”’1 CNN example with the MNIST dataframe
1.1 Baseline model
1.2 Simple Convolutional Neural Network for MNIST
1.3 Larger Convolutional Neural Network for MNIST
1.4 Improve model performance with image augmentation
1.4.1 Standardize pixel
1.4.2 ZCA Whitening
1.4.3 Random rotations
1.4.4 Random shifts
1.4.5 Random flips
1.4.6 Saving augmented images to file

2 Project object recognition in photographs
2.1 Simple CNN for CIFAR-10
2.2 Larger CNN for CIFAR-10
2.3 Extensions to improve model performance

3 Recurrent Neural Networks
3.1 Time Series Prediction with Multilayer Perceptrons (For comparisons)
3.1.1 Multilayer Perceptron Regression
3.1.2 Multilayer Perceptron Using the Window Method
3.2 Time series prediction
3.2.1 LSTM for Recurrent Neural Networks
3.2.2 LSTM for Regression using the Window method
3.2.3 LSTM With Memory Between Batches
3.2.4 Stacked LSTMs With Memory Between Batches
3.3 Example Sequence classification of movie reviews
3.3.1 LSTM for sequence classification with dropout
3.4 LSTM and CNN for sequence classification
3.5 Understanding stateful LSTM recurrent neural networks
3.5.1 LSTM for a Feature Window to One-Char Mapping
3.5.2 LSTM for a Time Step Window to One-Char Mapping
3.5.3 LSTM state maintained between samples within a batch
3.5.4 Stateful LSTM for a One-Char to One-Char Mapping
3.5.5 LSTM with variable length input to One-Char output
3.6 Example: Text generation with Alice in Wonderland
3.6.1 Develop a small LSTM recurrent neural network
3.6.2 Generating text with an LSTM network
3.6.3 Larger LSTM recurrent neural network (to improve results)

"""

"""1 CNN example with the MNIST dataframe"""
"""Handwritten digit classification problem. 28 x 28 pixel square (784 pixels total).
60,000 images are used to train a model and a separate set of 10,000 images are used to
test it."""
# Plot ad hoc mnist instances
from keras.datasets import mnist
import matplotlib.pyplot as plt
# Load the MNIST dataset

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# plot 4 images as gray scale
plt.subplot(221)
plt.imshow(X_train[0], cmap=plt.get_cmap( 'gray' ))
plt.subplot(222)
plt.imshow(X_train[1], cmap=plt.get_cmap( 'gray' ))
plt.subplot(223)
plt.imshow(X_train[2], cmap=plt.get_cmap( 'gray' ))
plt.subplot(224)
plt.imshow(X_train[3], cmap=plt.get_cmap( 'gray' ))
# show the plot
plt.show()

"""1.1 Baseline model"""
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.utils import np_utils
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# flatten 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype( 'float32' )
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype( 'float32' )
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# One hot encode outputs since the output is an integer from 0 to 9;
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# define the baseline model
def baseline_model():
# create model
    model = Sequential()
    model.add(Dense(num_pixels, input_dim=num_pixels, init= 'normal' , activation= 'relu' ))
    model.add(Dense(num_classes, init= 'normal' , activation= 'softmax' ))#Hidden layer
# Compile model
    model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
    return model
#Visible layer 784 -> Hidden layer 784 -> Output layer 10
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
verbose=0)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100)) #Baseline Error: 1.86%
```

```
"""1.2 Simple Convolutional Neural Network for MNIST"""
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
K.set_image_dim_ordering('th')
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][channels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype( 'float32' )
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype( 'float32' )
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
"""Network structure:
1. The first hidden layer is a convolutional layer called a Convolution2D. The layer
has 32 feature maps, which with the size of 5 x 5 and a rectifier activation function.
This is the input layer, expecting images with the structure outline above.
2. Next we define a pooling layer that takes the maximum value called MaxPooling2D. It
is configured with a pool size of 2 x 2.
3. The next layer is a regularization layer using dropout called Dropout. It is
configured to randomly exclude 20% of neurons in the layer in order to reduce
overfitting.
4. Next is a layer that converts the 2D matrix data to a vector called Flatten. It
allows the output to be processed by standard fully connected layers.
5. Next a fully connected layer with 128 neurons and rectifier activation function is
used.
6. Finally, the output layer has 10 neurons for the 10 classes and a softmax activation
 function to output probability-like predictions for each class."""
def baseline_model():
# create model
    model = Sequential()
    model.add(Convolution2D(32, 5, 5, border_mode= 'valid' , input_shape=(1, 28, 28), activation=
'relu' ))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation= 'relu' ))
```

```
    model.add(Dense(num_classes, activation= 'softmax' ))
# Compile model
    model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
    return model

# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
verbose=0)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("CNN Error: %.2f%%" % (100-scores[1]*100)) #CNN Error: 1.22%


"""1.3 Larger Convolutional Neural Network for MNIST"""
"""Network topology:
1. Convolutional layer with 30 feature maps of size 5 ⇥ 5.
2. Pooling layer taking the max over 2 ⇥ 2 patches.
3. Convolutional layer with 15 feature maps of size 3 ⇥ 3.
4. Pooling layer taking the max over 2 ⇥ 2 patches.
5. Dropout layer with a probability of 20%.
6. Flatten layer.
7. Fully connected layer with 128 neurons and rectifier activation.
8. Fully connected layer with 50 neurons and rectifier activation.
9. Output layer."""
# Larger CNN for the MNIST Dataset
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
K.set_image_dim_ordering( 'th' )
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype( 'float32' )
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype( 'float32' )
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
```

```
num_classes = y_test.shape[1]
# define the larger model
def larger_model():
# create model
    model = Sequential()
    model.add(Convolution2D(30, 5, 5, input_shape=(1, 28, 28), activation= 'relu' ))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Convolution2D(15, 3, 3, activation= 'relu' ))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation= 'relu' ))
    model.add(Dense(50, activation= 'relu' ))
    model.add(Dense(num_classes, activation= 'softmax' ))
# Compile model
    model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
    return model
# build the model
model = larger_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
verbose=0)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Large CNN Error: %.2f%%" % (100-scores[1]*100)) #Large CNN Error: 0.70%


"""1.4 Improve model performance with image augmentation"""
"""ImageDataGenerator class defines the configuration for image data preparation and
augmentation. The API is designed to be iterated by the deep learning model fitting
process, creating augmented image data for you just-in-time. This reduces your memory
overhead, but adds some additional time cost during model training.
Instead of calling the fit() function on our model, we must call the fit_generator()
function"""
# Plot of images as baseline for comparison
from keras.datasets import mnist
from matplotlib import pyplot
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# create a grid of 3x3 images
for i in range(0, 9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(X_train[i], cmap=pyplot.get_cmap( 'gray' ))

"""1.4.1 Standardize pixel values
It is also possible to standardize pixel values across the entire dataset. This is
called feature standardization and mirrors the type of standardization often performed
for each column in a tabular dataset."""
# Standardize images across the dataset, mean=0, stdev=1
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
```

```
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
# convert from int to float
X_train = X_train.astype( 'float32' )
X_test = X_test.astype( 'float32' )
# define data preparation
datagen = ImageDataGenerator(featurewise_center=True,
featurewise_std_normalization=True)#Feature normalization
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
# create a grid of 3x3 images
    for i in range(0, 9):
        pyplot.subplot(330 + 1 + i)
        pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap( 'gray' ))
# show the plot
    pyplot.show()
    break

"""1.4.2 ZCA Whitening
A whitening transform of an image is a linear algebra operation that reduces the
redundancy in the matrix of pixel images. Less redundancy in the image is intended to
better highlight the structures and features in the image to the learning algorithm."""
# ZCA whitening
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
# convert from int to float
X_train = X_train.astype( 'float32' )
X_test = X_test.astype( 'float32' )
# define data preparation
datagen = ImageDataGenerator(zca_whitening=True) #ZCA whitening
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
# create a grid of 3x3 images
    for i in range(0, 9):
        pyplot.subplot(330 + 1 + i)
        pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap( 'gray' ))
# show the plot
    pyplot.show()
    break
```

```
"""1.4.3 Random rotations
You can train your model to better handle rotations of images by artificially and
randomly rotating images from your dataset during training."""
# Random Rotations
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
# convert from int to float
X_train = X_train.astype( 'float32' )
X_test = X_test.astype( 'float32' )
# define data preparation
datagen = ImageDataGenerator(rotation_range=90)#Random rotation
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
# create a grid of 3x3 images
    for i in range(0, 9):
        pyplot.subplot(330 + 1 + i)
        pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap( 'gray' ))
# show the plot
    pyplot.show()
    break


"""1.4.4 Random shifts
You can train your deep learning network to expect and currently handle
off-center objects by artificially creating shifted versions of your training data."""
# Random Shifts
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
# convert from int to float
X_train = X_train.astype( 'float32' )
X_test = X_test.astype( 'float32' )
# define data preparation
shift = 0.2
datagen = ImageDataGenerator(width_shift_range=shift, height_shift_range=shift)#Random shifts
# fit parameters from data
datagen.fit(X_train)
```

```
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
# create a grid of 3x3 images
    for i in range(0, 9):
        pyplot.subplot(330 + 1 + i)
        pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap( 'gray' ))
# show the plot
    pyplot.show()
    break


"""1.4.5 Random flips
Keras supports random flipping along both the vertical and horizontal axes using the
vertical flip and horizontal flip arguments."""
# Random Flips
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
# convert from int to float
X_train = X_train.astype( 'float32' )
X_test = X_test.astype( 'float32' )
# define data preparation
datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
# create a grid of 3x3 images
    for i in range(0, 9):
        pyplot.subplot(330 + 1 + i)
        pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap( 'gray' ))
# show the plot
    pyplot.show()
    break


"""1.4.6 Saving augmented images to file
Keras allows you to save the images generated during training. The directory, filename
prefix and image file type can be specified to the flow() function before training."""
# Save augmented images to file
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
import os
from keras import backend as K
K.set_image_dim_ordering( 'th' )
# load data
```

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
# convert from int to float
X_train = X_train.astype( 'float32' )
X_test = X_test.astype( 'float32' )
# define data preparation
datagen = ImageDataGenerator()
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
os.makedirs( 'images' )
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9, save_to_dir= 'images' ,
save_prefix= 'aug' , save_format= 'png' ): #Save images
# create a grid of 3x3 images
    for i in range(0, 9):
        pyplot.subplot(330 + 1 + i)
        pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap( 'gray' ))
# show the plot
    pyplot.show()
    break


"""2 Project object recognition in photographs
Object recognition is when a model is able to identify objects in images.
The 60,000 photos of the dataframe are in color with red, green and blue channels, but
are small measuring 32 x 32 pixel squares. 10 classes"""
# Plot ad hoc CIFAR10 instances
from keras.datasets import cifar10
from matplotlib import pyplot
#Deprecated: from scipy.misc import toimage. Correction goes next:
from PIL import Image
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# create a grid of 3x3 images
for i in range(0, 9): #Example with 10 images
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(Image.fromarray(X_train[i]))
# show the plot
    pyplot.show()

"""2.1 Simple CNN for CIFAR-10"""
# Simple CNN model for CIFAR-10
import numpy
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.constraints import maxnorm
from keras.optimizers import SGD
```

```
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
K.set_image_dim_ordering( 'th' )
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# normalize inputs from 0-255 to 0.0-1.0
X_train = X_train.astype( 'float32' )
X_test = X_test.astype( 'float32' )
X_train = X_train / 255.0
X_test = X_test / 255.0
# one hot encode outputs: To transform output variables into a binary matrix in order to best model
the classification problem.
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
"""1. Convolutional input layer, 32 feature maps with a size of 3 x 3, a rectifier activation
function and a weight constraint of max norm set to 3.
2. Dropout set to 20%.
3. Convolutional layer, 32 feature maps with a size of 3 x 3, a rectifier activation function
and a weight constraint of max norm set to 3.
4. Max Pool layer with the size 2 x 2.
5. Flatten layer.
6. Fully connected layer with 512 units and a rectifier activation function.
7. Dropout set to 50%.
8. Fully connected output layer with 10 units and a softmax activation function."""
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# normalize inputs from 0-255 to 0.0-1.0
X_train = X_train.astype( 'float32' )
X_test = X_test.astype( 'float32' )
X_train = X_train / 255.0
X_test = X_test / 255.0
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# Create the model
model = Sequential()
model.add(Convolution2D(32, 3, 3, input_shape=(3, 32, 32), border_mode= 'same' ,
activation= 'relu' , W_constraint=maxnorm(3)))
model.add(Dropout(0.2))
model.add(Convolution2D(32, 3, 3, activation= 'relu' , border_mode= 'same' ,
W_constraint=maxnorm(3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
model.add(Flatten())
model.add(Dense(512, activation= 'relu' , W_constraint=maxnorm(3)))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation= 'softmax' ))
# Compile model
epochs = 25
lrate = 0.01
decay = lrate/epochs
sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss= 'categorical_crossentropy' , optimizer='sgd', metrics=[ 'accuracy' ])
print(model.summary())
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=epochs, batch_size=32,
verbose=0)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100)) #Accuracy: 69.76%


"""2.2 Larger CNN for CIFAR-10"""
"""1. Convolutional input layer, 32 feature maps with a size of 3 ⇥ 3 and a rectifier activation
function.
2. Dropout layer at 20%.
3. Convolutional layer, 32 feature maps with a size of 3 ⇥ 3 and a rectifier activation function.
4. Max Pool layer with size 2 ⇥ 2.
5. Convolutional layer, 64 feature maps with a size of 3 ⇥ 3 and a rectifier activation function.
6. Dropout layer at 20%.
7. Convolutional layer, 64 feature maps with a size of 3 ⇥ 3 and a rectifier activation function.
8. Max Pool layer with size 2 ⇥ 2.
9. Convolutional layer, 128 feature maps with a size of 3 ⇥ 3 and a rectifier activation function.
10. Dropout layer at 20%.
11. Convolutional layer, 128 feature maps with a size of 3 ⇥ 3 and a rectifier activation function.
12. Max Pool layer with size 2 ⇥ 2.
13. Flatten layer.
14. Dropout layer at 20%.
15. Fully connected layer with 1,024 units and a rectifier activation function.
16. Dropout layer at 20%.
17. Fully connected layer with 512 units and a rectifier activation function.
18. Dropout layer at 20%.
19. Fully connected output layer with 10 units and a softmax activation function."""
# Large CNN model for the CIFAR-10 Dataset
import numpy
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.constraints import maxnorm
from keras.optimizers import SGD
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
```

```
from keras import backend as K
K.set_image_dim_ordering( 'th' )
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# normalize inputs from 0-255 to 0.0-1.0
X_train = X_train.astype( 'float32' )
X_test = X_test.astype( 'float32' )
X_train = X_train / 255.0
X_test = X_test / 255.0
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# Create the model
model = Sequential()
model.add(Convolution2D(32, 3, 3, input_shape=(3, 32, 32), activation= 'relu' ,
border_mode= 'same' ))
model.add(Dropout(0.2))
model.add(Convolution2D(32, 3, 3, activation= 'relu' , border_mode= 'same' ))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Convolution2D(64, 3, 3, activation= 'relu' , border_mode= 'same' ))
model.add(Dropout(0.2))
model.add(Convolution2D(64, 3, 3, activation= 'relu' , border_mode= 'same' ))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Convolution2D(128, 3, 3, activation= 'relu' , border_mode= 'same' ))
model.add(Dropout(0.2))
model.add(Convolution2D(128, 3, 3, activation= 'relu' , border_mode= 'same' ))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(1024, activation= 'relu' , W_constraint=maxnorm(3)))
model.add(Dropout(0.2))
model.add(Dense(512, activation= 'relu' , W_constraint=maxnorm(3)))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation= 'softmax' ))
# Compile model
epochs = 25
lrate = 0.01
decay = lrate/epochs
sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss= 'categorical_crossentropy' , optimizer='sgd', metrics=[ 'accuracy' ])
print(model.summary())
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=epochs, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100)) #Accuracy: 78.77%
```

"""2.3 Extensions to improve model performance"""
"""- More epochs
- Data augmentation
- Deeper network topology"""

"""3 Recurrent Neural Networks"""
"""Recurrent neural networks have connections that have loops, adding feedback and memory to the networks over time. This memory allows this type of network to learn and generalize across sequences of inputs rather than individual patterns. For sequence problems.
Long Short-Term Memory Network (LSTM) is impressive.
Taxonomy of sequence problems that require a mapping of an input to an output:
- One-to-Many: sequence output, for image captioning.
- Many-to-One: sequence input, for sentiment classification.
- Many-to-Many: sequence in and out, for machine translation.
- Synchronized Many-to-Many: synced sequences in and out, for video classification."""

"""How to Train Recurrent Neural Networks: Back propagation breaks down in a recurrent neural network, because of the recurrent or loop connections.
How to Have Stable Gradients During Training: When Back propagation is used in very deep neural networks and in unrolled recurrent neural
networks, the gradients that are calculated in order to update the weights can become unstable. They can become very large numbers called exploding gradients or very small numbers called the vanishing gradient problem. These large numbers in turn are used to update the weights in the network, making training unstable and the network unreliable.
"""

"""Long Short-Term Memory or LSTM network is a recurrent neural network that is trained using Back propagation Through Time and overcomes the vanishing gradient problem.
Instead of neurons, LSTM networks have memory blocks that are connected into layers.
A block has components that make it smarter than a classical neuron and a memory for recent sequences. A block contains gates that manage the block's state and output. A unit operates upon an input sequence and each gate within a unit uses the sigmoid activation function to control whether they are triggered or not, making the change of state and addition of information flowing through the unit conditional. There are three types of gates within a memory unit:
- Forget Gate: conditionally decides what information to discard from the unit.
- Input Gate: conditionally decides which values from the input to update the memory state.
- Output Gate: conditionally decides what to output based on input and the memory of the unit.
Each unit is like a mini state machine where the gates of the units have weights that are learned during the training procedure."""

"""3.1 Time Series Prediction with Multilayer Perceptrons (For comparisons)"""
"""The task is to predict the number of international airline passengers in units of 1,000."""
#Download the dataframe
import csv
import requests

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv'
response = requests.get(url)
dataframe = pandas.read_csv( url , usecols=[1], engine= 'python' , skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype( 'float32' )
plt.plot(dataset)
dataframe.to_csv('dataframe.csv')
dataframe=pd.read_csv('dataframe.csv')

"""3.1.1 Multilayer Perceptron Regression"""
import numpy
import matplotlib.pyplot as plt
import pandas
import math
from keras.models import Sequential
from keras.layers import Dense
# fix random seed for reproducibility
numpy.random.seed(7)

# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
print(len(train), len(test))

# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1): #look back which is the number of previous time steps
to use as input variables to predict the next time period, in this case, defaulted to 1
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
        return numpy.array(dataX), numpy.array(dataY)

#Prepare the train and test datafranes for modelling: reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)

# create and fit Multilayer Perceptron model
model = Sequential()
model.add(Dense(8, input_dim=look_back, activation= 'relu' ))
model.add(Dense(1))
model.compile(loss= 'mean_squared_error' , optimizer= 'adam' )
model.fit(trainX, trainY, nb_epoch=200, batch_size=2, verbose=2)

# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
```

```
print( 'Train Score: %.2f MSE (%.2f RMSE)' % (trainScore, math.sqrt(trainScore))) #Train Score: 6462.03 MSE (80.39 RMSE)
testScore = model.evaluate(testX, testY, verbose=0)
print( 'Test Score: %.2f MSE (%.2f RMSE)' % (testScore, math.sqrt(testScore))) #Test Score: 46438.42 MSE (215.50 RMSE)

# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()


"""3.1.2 Multilayer Perceptron Using the Window Method"""
# Multilayer Perceptron to Predict International Airline Passengers (t+1, given t, t-1, t-2)
import numpy
import matplotlib.pyplot as plt
import pandas
import math
from keras.models import Sequential
from keras.layers import Dense
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
            a = dataset[i:(i+look_back), 0]
            dataX.append(a)
            dataY.append(dataset[i + look_back, 0])
            return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
print(len(train), len(test))
# reshape dataset
look_back = 10
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# create and fit Multilayer Perceptron model
```

```
model = Sequential()
model.add(Dense(8, input_dim=look_back, activation= 'relu' ))
model.add(Dense(1))
model.compile(loss= 'mean_squared_error' , optimizer= 'adam' )
model.fit(trainX, trainY, nb_epoch=200, batch_size=2, verbose=2)
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print( 'Train Score: %.2f MSE (%.2f RMSE)' % (trainScore, math.sqrt(trainScore))) #Train Score:
1193.59 MSE (34.55 RMSE)
testScore = model.evaluate(testX, testY, verbose=0)
print( 'Test Score: %.2f MSE (%.2f RMSE)' % (testScore, math.sqrt(testScore))) #Test Score:
45324.43 MSE (212.90 RMSE)
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()


"""3.2 Time series prediction with LSTM"""
"""3.2.1 LTSM for Recurrent Neural Networks"""
"""LSTM network is a type of recurrent neural network used in deep learning because
very large architectures can be successfully trained.
LSTMs are sensitive to the scale of the input data, specifically when the sigmoid
(default) or tanh activation functions are used. It can be a good practice to rescale
the data to the range of 0-to-1, also called normalizing."""
# LSTM for international airline passengers problem with regression framing
import numpy
import matplotlib.pyplot as plt
import pandas
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
```

```
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
        return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = pandas.read_csv( 'dataframe.csv' , usecols=[1],
engine= 'python' , skipfooter=3)#I do not have it in my laptop
dataset = dataframe.values
dataset = dataset.astype( 'float32' )
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_dim=look_back))
model.add(Dense(1))
model.compile(loss= 'mean_squared_error' , optimizer= 'adam' )
model.fit(trainX, trainY, nb_epoch=100, batch_size=1, verbose=2)
# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print( 'Train Score: %.2f RMSE' % (trainScore)) #Train Score: 0.06 RMSE
testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print( 'Test Score: %.2f RMSE' % (testScore)) #Test Score: 173.97 RMSE
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
```

```
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()


"""3.2.2 LSTM for Regression using the Window method"""
# LSTM for international airline passengers problem with window regression framing
import numpy
import matplotlib.pyplot as plt
import pandas
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
        return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = pandas.read_csv( 'dataframe.csv' , usecols=[1], engine= 'python' , skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype( 'float32' )
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_dim=look_back))
model.add(Dense(1))
model.compile(loss= 'mean_squared_error' , optimizer= 'adam' )
model.fit(trainX, trainY, nb_epoch=100, batch_size=1, verbose=2)
# make predictions
```

```
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0])) #Train Score: 0.04 RMSE
print( 'Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0])) #Test Score: 109.51 RMSE
print( 'Test Score: %.2f RMSE' % (testScore))
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()


"""3.2.3 LSTM For Regression with Time Steps"""
"""We can do this using the same data representation as in the previous window-based
example, except when we reshape the data we set the columns to be the time steps
dimension and change the features dimension back to 1."""
# LSTM for international airline passengers problem with time step regression framing
import numpy
import matplotlib.pyplot as plt
import pandas
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
        return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
```

```
dataframe = pandas.read_csv( 'dataframe.csv' , usecols=[1],
engine= 'python' , skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype( 'float32' )
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_dim=1))
model.add(Dense(1))
model.compile(loss= 'mean_squared_error' , optimizer= 'adam' )
model.fit(trainX, trainY, nb_epoch=100, batch_size=1, verbose=2)
# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print( 'Train Score: %.2f RMSE' % (trainScore)) #Train Score: 58689.59 RMSE
testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print( 'Test Score: %.2f RMSE' % (testScore)) #Test Score: 89199.80 RMSE
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

```
"""3.2.3 LSTM With Memory Between Batches"""
"""We can gain finer control over when the internal state of the LSTM network is
cleared in Keras by making the LSTM layer stateful. This means that it can build state
over the entire training sequence and even maintain that state if needed to make
predictions. It requires that the training data not be shu✏️ed when fitting the network.
 It also requires explicit resetting of the network state after each exposure to the
 training data (epoch) by calls to model.reset states(). This means that we must create
 our own outer loop of epochs and within each epoch call model.fit() and model.reset
 states()"""
# LSTM for international airline passengers problem with memory
import numpy
import matplotlib.pyplot as plt
import pandas
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
        return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = pandas.read_csv( 'dataframe.csv' , usecols=[1], engine= 'python' , skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype( 'float32' )
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))
# create and fit the LSTM network
batch_size = 1
model = Sequential()
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True))
model.add(Dense(1))
```

```
model.compile(loss= 'mean_squared_error' , optimizer= 'adam' )
for i in range(100):
    model.fit(trainX, trainY, nb_epoch=1, batch_size=batch_size, verbose=2, shuffle=False)
    model.reset_states()
# make predictions
trainPredict = model.predict(trainX, batch_size=batch_size)
model.reset_states()
testPredict = model.predict(testX, batch_size=batch_size)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print( 'Train Score: %.2f RMSE' % (trainScore)) #Train Score: 0.07 RMSE
testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print( 'Test Score: %.2f RMSE' % (testScore)) #Test Score: 148.69 RMSE
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
###trainPredictPlot[:, :] = numpy.nan It gives me an error
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
###testPredictPlot[:, :] = numpy.nan It gives me an error
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()


"""3.2.4 Stacked LSTMs With Memory Between Batches"""
"""It is required is that an LSTM layer prior to each subsequent LSTM layer must
return the sequence. This can be done by setting the return sequences parameter on the
 layer to True."""
# Stacked LSTM for international airline passengers problem with memory
import numpy
import matplotlib.pyplot as plt
import pandas
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
```

```
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
        return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = pandas.read_csv( 'dataframe.csv' , usecols=[1], engine= 'python' , skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype( 'float32' )
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))
# create and fit the LSTM network
batch_size = 1
model = Sequential()
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True,
return_sequences=True))
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True))
model.add(Dense(1))
model.compile(loss= 'mean_squared_error' , optimizer= 'adam' )
for i in range(100):
    model.fit(trainX, trainY, nb_epoch=1, batch_size=batch_size, verbose=2, shuffle=False)
    model.reset_states()
# make predictions
trainPredict = model.predict(trainX, batch_size=batch_size)
model.reset_states()
testPredict = model.predict(testX, batch_size=batch_size)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print( 'Train Score: %.2f RMSE' % (trainScore)) #Train Score: 0.14 RMSE
testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print( 'Test Score: %.2f RMSE' % (testScore)) #Test Score: 169.01 RMSE
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
```

```
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()


"""3.3 Example Sequence classification of movie reviews"""
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset but only keep the top n words, zero the rest. We are constraining the dataset to the top 5,000 words.
#We also split the dataset into train (50%) and test (50%) sets.
top_words = 5000
# I must do this trick to avoid an error
numpy.load.__defaults__=(None, True, True, 'ASCII')
numpy_load_old = numpy.load
numpy.load = lambda *a,**k: numpy_load_old(*a, allow_pickle=True, **k)
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
numpy.load = numpy_load_old
(X_train, y_train), (X_test, y_test) = imdb.load_data()
X = numpy.concatenate((X_train, X_test), axis=0)
y = numpy.concatenate((y_train, y_test), axis=0)
numpy.load.__defaults__=(None, False, True, 'ASCII')
# truncate and pad input sequences, so that sequences have the same length
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
#I use top_words = 250000 since otherwise it gives me an error
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation= 'sigmoid' ))
model.compile(loss= 'binary_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
print(model.summary())
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100)) #Accuracy: 85.87%
```

```
"""3.3.1 LSTM for sequence classification with dropout"""
"""Recurrent Neural networks like LSTM generally have the problem of overfitting.
Dropout can be applied between layers using the Dropout Keras layer. We can do this
easily by adding new Dropout layers between the Embedding and LSTM layers and the LSTM
and Dense output layers. We can also add dropout to the input on the Embedded layer by
using the dropout parameter."""
# LSTM with Dropout for sequence classification in the IMDB dataset
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
# I must do this trick to avoid an error
numpy.load.__defaults__=(None, True, True, 'ASCII')
numpy_load_old = numpy.load
numpy.load = lambda *a,**k: numpy_load_old(*a, allow_pickle=True, **k)
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
numpy.load = numpy_load_old
(X_train, y_train), (X_test, y_test) = imdb.load_data()
X = numpy.concatenate((X_train, X_test), axis=0)
y = numpy.concatenate((y_train, y_test), axis=0)
numpy.load.__defaults__=(None, False, True, 'ASCII')
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
#I use top_words = 250000 since otherwise it gives me an error
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length,
dropout=0.2))
model.add(Dropout(0.2))
model.add(LSTM(100))
model.add(Dropout(0.2))
model.add(Dense(1, activation= 'sigmoid' ))
model.compile(loss= 'binary_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
print(model.summary())
model.fit(X_train, y_train, epochs=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100)) #Accuracy: 86.34%
```

"""Alternately, dropout can be applied to the input and recurrent connections of the
memory units with the LSTM precisely and separately. Keras provides this capability
with parameters on the LSTM layer, the dropout_W for configuring the input dropout and
 dropout_U for configuring the recurrent dropout."""
# LSTM with dropout for sequence classification in the IMDB dataset
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
# I must do this trick to avoid an error
numpy.load.__defaults__=(None, True, True, 'ASCII')
numpy_load_old = numpy.load
numpy.load = lambda *a,**k: numpy_load_old(*a, allow_pickle=True, **k)
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
numpy.load = numpy_load_old
(X_train, y_train), (X_test, y_test) = imdb.load_data()
X = numpy.concatenate((X_train, X_test), axis=0)
y = numpy.concatenate((y_train, y_test), axis=0)
numpy.load.__defaults__=(None, False, True, 'ASCII')
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=top_words)
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
#I use top_words = 250000 since otherwise it gives me an error
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length,
dropout=0.2))
model.add(LSTM(100, dropout_W=0.2, dropout_U=0.2))
model.add(Dense(1, activation= 'sigmoid'))
model.compile(loss= 'binary_crossentropy' , optimizer= 'adam' , metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, nb_epoch=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100)) #Accuracy: 83.30%

"""We can see that the LSTM specific dropout has a more pronounced e↩ect on the
convergence of the network than the layer-wise dropout. As above, the number of epochs
was kept constant and could be increased to see if the skill of the model can be
further lifted."""

```
"""3.4 LSTM and CNN for sequence classification"""
"""Convolutional neural networks excel at learning the spatial structure in input data.
 The IMDB review data does have a one-dimensional spatial structure in the sequence of
 words in reviews and the CNN may be able to pick out invariant features for good and
 bad sentiment. This learned spatial features may then be learned as sequences by an
 LSTM layer. We can easily add a one-dimensional CNN and max pooling layers after the
 Embedding layer which then feed the consolidated features to the LSTM."""
# LSTM and CNN for sequence classification in the IMDB dataset
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.convolutional import Convolution1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
# I must do this trick to avoid an error
numpy.load.__defaults__=(None, True, True, 'ASCII')
numpy_load_old = numpy.load
numpy.load = lambda *a,**k: numpy_load_old(*a, allow_pickle=True, **k)
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=top_words)
numpy.load = numpy_load_old
(X_train, y_train), (X_test, y_test) = imdb.load_data()
X = numpy.concatenate((X_train, X_test), axis=0)
y = numpy.concatenate((y_train, y_test), axis=0)
numpy.load.__defaults__=(None, False, True, 'ASCII')
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
#I use top_words = 250000 since otherwise it gives me an error
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(Convolution1D(nb_filter=32, filter_length=3, border_mode= 'same' ,
activation= 'relu' ))
model.add(MaxPooling1D(pool_length=2))
model.add(LSTM(100))
model.add(Dense(1, activation= 'sigmoid' ))
model.compile(loss= 'binary_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
print(model.summary())
model.fit(X_train, y_train, epochs=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100)) # Accuracy: 86.00%
```

```
"""We can see that we achieve similar results to the first example although with less
weights and faster training time."""


"""3.5 Understanding stateful LSTM recurrent neural networks"""
"""Problem description: Learn the alphabet. Given a letter of the alphabet, predict the
 next letter of the alphabet. A classification problem"""
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
# fix random seed for reproducibility
numpy.random.seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
    seq_in = alphabet[i:i + seq_length]
    seq_out = alphabet[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
    print(seq_in, '->' , seq_out)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (len(dataX), seq_length, 1))
# normalize the input integers to the range 0-to-1, the range of the sigmoid activation functions
X = X / float(len(alphabet))
"""We can think of this problem as a sequence classification task, where each of the
26 letters represents a di↵erent class. As such, we can convert the output (y) to a
one hot encoding, using the Keras built-in function to categorical()."""
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
"""We will frame the problem as a random collection of one-letter input to one-letter
output pairs.Let’s define an LSTM network with 32 units and a single output neuron
with a softmax activation function for making predictions. Because this is a multiclass
classification problem, we can use the log loss function (called categorical
crossentropy in Keras), and optimize the network using the ADAM optimization function.
The model is fit over 500 epochs with a batch size of 1."""
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation= 'softmax' ))
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
model.fit(X, y, epochs=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
```

```
print("Model Accuracy: %.2f%%" % (scores[1]*100)) #Model Accuracy: 92.00%
# demonstrate some model predictions
for pattern in dataX:
    x = numpy.reshape(pattern, (1, len(pattern), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print (seq_in, "->", result)
"""We can see that this problem is indeed difficult for the network to learn. The
reason is, the poor LSTM units do not have any context to work with. Each input-output
 pattern is shown to the network in a random order and the state of the network is
 reset after each pattern (each batch where each batch contains one pattern)."""


"""3.5.1 LSTM for a Feature Window to One-Char Mapping"""
"""A popular approach to adding more context to data for Multilayer Perceptrons is to
use the window method. This is where previous steps in the sequence are provided as
additional input features to the network."""
# Naive LSTM to learn three-char window to one-char mapping
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
# fix random seed for reproducibility
numpy.random.seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 3 #I.e. ABC -> D
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
    seq_in = alphabet[i:i + seq_length]
    seq_out = alphabet[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
    print (seq_in, '->' , seq_out)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (len(dataX), 1, seq_length))#Each element in the sequence is then
provided as a new input feature to the network.
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
model = Sequential()
```

```
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation= 'softmax' ))
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
model.fit(X, y, epochs=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100)) #Model Accuracy: 86.96%
# demonstrate some model predictions
for pattern in dataX:
    x = numpy.reshape(pattern, (1, 1, len(pattern))) #It also requires a modification for how the
sample patterns are reshaped when demonstrating predictions from the model.
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print (seq_in, "->", result)


"""3.5.2 LSTM for a Time Step Window to One-Char Mapping"""
# Naive LSTM to learn three-char time steps to one-char mapping
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
# fix random seed for reproducibility
numpy.random.seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 3 #I.e. ABC -> D
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
    seq_in = alphabet[i:i + seq_length]
    seq_out = alphabet[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
    print (seq_in, '->' , seq_out)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (len(dataX), seq_length, 1))#The di↩erence is that the reshaping of the
input data takes the sequence as a time step sequence of one feature, rather than a single time step of
multiple features.
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
```

```
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation= 'softmax' ))
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
model.fit(X, y, epochs=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100)) #Model Accuracy: 100.00% Perfection!
# demonstrate some model predictions
for pattern in dataX:
    x = numpy.reshape(pattern, (1, len(pattern), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print (seq_in, "->", result)


"""3.5.3 LSTM state maintained between samples within a batch"""
"""The Keras implementation of LSTMs resets the state of the network after each batch.
Keras shuffles the training dataset before each training epoch."""
# Naive LSTM to learn one-char to one-char mapping with all data in each batch
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
from keras.preprocessing.sequence import pad_sequences
# fix random seed for reproducibility
numpy.random.seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
    seq_in = alphabet[i:i + seq_length]
    seq_out = alphabet[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
    print( seq_in, "->" , seq_out)
# convert list of lists to array and pad sequences if needed
X = pad_sequences(dataX, maxlen=seq_length, dtype= 'float32' )
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (X.shape[0], seq_length, 1))
# normalize
X = X / float(len(alphabet))
```

```
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(16, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation= 'softmax' ))
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
model.fit(X, y, nb_epoch=5000, batch_size=len(dataX), verbose=2, shuffle=False) #shuffle = False
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100)) #Model Accuracy: 100.00%
# Demonstrate some model predictions
for pattern in dataX:
    x = numpy.reshape(pattern, (1, len(pattern), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print (seq_in, "->", result)
# Demonstrate predicting random patterns
print ("Test a Random Pattern:")
for i in range(0,20):
    pattern_index = numpy.random.randint(len(dataX))
    pattern = dataX[pattern_index]
    x = numpy.reshape(pattern, (1, len(pattern), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print (seq_in, "->", result)


"""3.5.4 Stateful LSTM for a One-Char to One-Char Mapping"""
"""We want to expose the network to the entire sequence and let it learn the
inter-dependencies, rather than us define those dependencies explicitly in the framing
of the problem. We can do this in Keras by making the LSTM layers stateful and manually
 resetting the state of the network at the end of the epoch, which is also the end of
 the training sequence.
An important difference in training the stateful LSTM is that we train it manually one
epoch at a time and reset the state after each epoch. We can do this in a for loop.
Again, we do not shuffle the input, preserving the sequence in which the input training
 data was created."""
# Stateful LSTM to learn one-char to one-char mapping
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
# fix random seed for reproducibility
numpy.random.seed(7)
```

```
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
    seq_in = alphabet[i:i + seq_length]
    seq_out = alphabet[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
    print (seq_in, "->" , seq_out)
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (len(dataX), seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
batch_size = 1
model = Sequential()
model.add(LSTM(16, batch_input_shape=(batch_size, X.shape[1], X.shape[2]), stateful=True))
model.add(Dense(y.shape[1], activation= 'softmax' ))
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
for i in range(300):
    model.fit(X, y, epochs=1, batch_size=batch_size, verbose=2, shuffle=False)
    model.reset_states()
# summarize performance of the model
scores = model.evaluate(X, y, batch_size=batch_size, verbose=0) #we specify the batch size when
evaluating the performance of the network on the entire training dataset.
model.reset_states()
print("Model Accuracy: %.2f%%" % (scores[1]*100)) #Model Accuracy: 36.00%
# demonstrate some model predictions
seed = [char_to_int[alphabet[0]]]
for i in range(0, len(alphabet)-1):
    x = numpy.reshape(seed, (1, len(seed), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    print (int_to_char[seed[0]], "->", int_to_char[index])
seed = [index]
model.reset_states()
# demonstrate a random starting point
letter = "K"
seed = [char_to_int[letter]]
print ("New start: ", letter)
for i in range(0, 5):
    x = numpy.reshape(seed, (1, len(seed), 1))
    x = x / float(len(alphabet))
```

```
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    print (int_to_char[seed[0]], "->", int_to_char[index])
seed = [index]
model.reset_states()


"""3.5.5 LSTM with variable length input to One-Char output"""
"""We explore a variation of the stateless LSTM that learns random subsequences of
the alphabet and an e↩ort to build a model that can be given arbitrary letters or
subsequences of letters and predict the next letter in the alphabet."""
"""Firstly, we are changing the framing of the problem. To simplify we will define a maximum
input sequence length and set it to a small value like 5 to speed up training. This defines the
maximum length of subsequences of the alphabet will be drawn for training. In extensions, this
could just as set to the full alphabet (26) or longer if we allow looping back to the start of the
sequence."""
# LSTM with Variable Length Input Sequences to One Character Output
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
from keras.preprocessing.sequence import pad_sequences
# fix random seed for reproducibility
numpy.random.seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
num_inputs = 1000
max_len = 5
dataX = []
dataY = []
for i in range(num_inputs):
    start = numpy.random.randint(len(alphabet)-2)
    end = numpy.random.randint(start, min(start+max_len,len(alphabet)-1))
    sequence_in = alphabet[start:end+1]
    sequence_out = alphabet[end + 1]
    dataX.append([char_to_int[char] for char in sequence_in])
    dataY.append(char_to_int[sequence_out])
    print (sequence_in, "->" , sequence_out)
# convert list of lists to array and pad sequences if needed
X = pad_sequences(dataX, maxlen=max_len, dtype= 'float32' )
# reshape X to be [samples, time steps, features]
X = numpy.reshape(X, (X.shape[0], max_len, 1))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
```

```
batch_size = 1
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], 1)))
model.add(Dense(y.shape[1], activation= 'softmax' ))
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
model.fit(X, y, epochs=500, batch_size=batch_size, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100)) #Model Accuracy: 98.50%
# demonstrate some model predictions
for i in range(20):
    pattern_index = numpy.random.randint(len(dataX))
    pattern = dataX[pattern_index]
    x = pad_sequences([pattern], maxlen=max_len, dtype= 'float32' )
    x = numpy.reshape(x, (1, max_len, 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print (seq_in, "->", result)


"""3.6 Example: Text generation with Alice in Wonderland"""
"""Recurrent neural networks can also be used as generative models. This means that in
 addition to being used for predictive models they can learn the sequences of a
problem and then generate entirely new plausible sequences for the problem domain."""

"""3.6.1 Develop a small LSTM recurrent neural network"""
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils

# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()#Load the example book
raw_text = raw_text.lower()

# create mapping of unique chars to integers, and a reverse mapping
"""We cannot model the characters directly, instead we must convert the characters to
integers. We can do this easily by first creating a set of all of the distinct
characters in the book, then creating a map of each character to a unique integer."""
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))

# Summarize the dataframe
n_chars = len(raw_text)
n_vocab = len(chars)
```

```
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)

"""We will split the book text up into subsequences with a fixed length of 100
characters, an arbitrary length. We could just as easily split the data up by
sentences and pad the shorter sequences and truncate the longer ones.
Each training pattern of the network is comprised of 100 time steps of one character
(X) followed by one character output (y). When creating these sequences, we slide this
window along the whole book one character at a time, allowing each character a chance
to be learned from the 100 characters that preceded it (except the first 100
characters of course)."""
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print ("Total Patterns: ", n_patterns) #Total Patterns:  144331

"""Now that we have prepared our training data we need to transform it so that it is
suitable for use with Keras. First we must transform the list of input sequences into
the form [samples, time steps, features] expected by an LSTM network. Next we need to
rescale the integers to the range 0-to-1 to make the patterns easier to learn by the
LSTM network that uses the sigmoid activation function by default.
Finally, we need to convert the output patterns (single characters converted to
integers) into a one hot encoding. This is so that we can configure the network to
predict the probability of each of the 47 different characters in the vocabulary (an
easier representation) rather than trying to force it to predict precisely the next
character. Each y value is converted into a sparse vector with a length of 47, full of
zeros except with a 1 in the column for the letter (integer) that the pattern
represents."""
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)

"""We can now define our LSTM model. Here we define a single hidden LSTM layer with
256 memory units. The network uses dropout with a probability of 20%. The output layer
 is a Dense layer using the softmax activation function to output a probability
 prediction for each of the 47 characters between 0 and 1. The problem is really a
 single character classification problem with 47 classes and as such is defined as
 optimizing the log loss (cross entropy), here using the ADAM optimization algorithm
 for speed."""
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
```

```
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation= 'softmax' ))
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' )

"""There is no test dataset. We are modeling the entire training dataset to learn the
probability of each character in a sequence. We are not interested in the most
accurate (classification accuracy) model of the training dataset. This would be a
model that predicts each character in the training dataset perfectly. Instead we are
interested in a generalization of the dataset that minimizes the chosen loss function.
We are seeking a balance between generalization and overfitting but short of
memorization."""
"""We will use model checkpointing to record all of the network weights to file each
time an improvement in loss is observed at the end of the epoch. We will use the best
set of weights (lowest loss) to instantiate our generative model in the next section."""
# define the checkpoint
filepath="weights-improvement-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor= 'loss' , verbose=1, save_best_only=True, mode=
min )
callbacks_list = [checkpoint]
#Fit the model
model.fit(X, y, epochs=20, batch_size=128, callbacks=callbacks_list)

from keras.models import load_model

model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
del model  # deletes the existing model

# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')



"""3.6.2 Generating text with an LSTM network"""
# Load LSTM network and generate text
import sys
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers, and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
int_to_char = dict((i, c) for i, c in enumerate(chars)) """We must also create a
reverse mapping that we can use to convert the integers back to characters so that we
```

```
can understand the predictions."""
# summarize the loaded data
n_chars = len(raw_text)
n_vocab = len(chars)
print ("Total Characters: ", n_chars) #Total Characters:  144431
print ("Total Vocab: ", n_vocab) #Total Vocab:  45
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print ("Total Patterns: ", n_patterns) #Total Patterns:  144331
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation= 'softmax' ))
# load the network weights
filename = "weights-improvement-16-1.8732.hdf5"#The best epoch in the previous point
model.load_weights(filename)
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' )
# pick a random seed
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print ("Seed:")
print ("\"", ".join([int_to_char[value] for value in pattern]), "\"")
"""The simplest way to use the Keras LSTM model to make predictions is to first start
off with a seed sequence as input, generate the next character then update the seed
sequence to add the generated character on the end and trim off the first character.
This process is repeated for as long as we want to predict new characters (e.g. a
sequence of 1,000 characters in length). We can pick a random input pattern as our
seed sequence, then print generated characters as we generate them."""
# generate characters
for i in range(1000):
    x = numpy.reshape(pattern, (1, len(pattern), 1))
    x = x / float(n_vocab)
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    sys.stdout.write(result)
```

```
    pattern.append(index)
    pattern = pattern[1:len(pattern)]
print ("\nDone.")
"""We can note some observations about the generated text.
It generally conforms to the line format observed in the original text of less than 80 characters before a new line.
The characters are separated into word-like groups and most groups are actual English words (e.g. the, little and was), but many do not (e.g. lott, tiie and taede).
Some of the words in sequence make sense(e.g. and the white rabbit), but many do not (e.g. wese tilel )."""


"""3.6.3 Larger LSTM recurrent neural network (to improve results)"""
"""We will keep the number of memory units the same at 256, but add a second layer. We will also change the filename of the checkpointed weights so that we can tell the difference between weights for this network and the previous (by appending the word bigger in the filename). we will increase the number of training epochs from 20 to 50 and decrease the batch size from 128 to 64 to give the network more of an opportunity to be updated and learn. """
# Larger LSTM Network to Generate Text for Alice in Wonderland
import numpy
import sys
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
int_to_char = dict((i, c) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
n_vocab = len(chars)
print ("Total Characters: ", n_chars) #Total Characters:  144431
print ("Total Vocab: ", n_vocab) #Total Vocab:  45
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print ("Total Patterns: ", n_patterns) #Total Patterns:  144331
```

```
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation= 'softmax' ))
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' )
# define the checkpoint
filepath="weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor= 'loss' , verbose=1, save_best_only=True,
mode= min )
callbacks_list = [checkpoint]
# fit the model
model.fit(X, y, epochs=50, batch_size=64, callbacks=callbacks_list)
"""This step takes too much time. Just to test, I am going to stop it before it ends,
and select the best epoch so far to continue with the project."""

"""We can use this best model from the run to generate text. The only change we need
to make to the text generation script from the previous section is in the specification
 of the network topology and from which file to seed the network weights."""
# load the network weights
filename = "weights-improvement-03-2.2290-bigger.hdf5" #The best epoch
model.load_weights(filename)
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' )
# pick a random seed
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print ("Seed:")
print ("\"",".join([int_to_char[value] for value in pattern]), "\"")
# generate characters
for i in range(1000):
    x = numpy.reshape(pattern, (1, len(pattern), 1))
    x = x / float(n_vocab)
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    sys.stdout.write(result)
    pattern.append(index)
    pattern = pattern[1:len(pattern)]
print ("\nDone.")
```