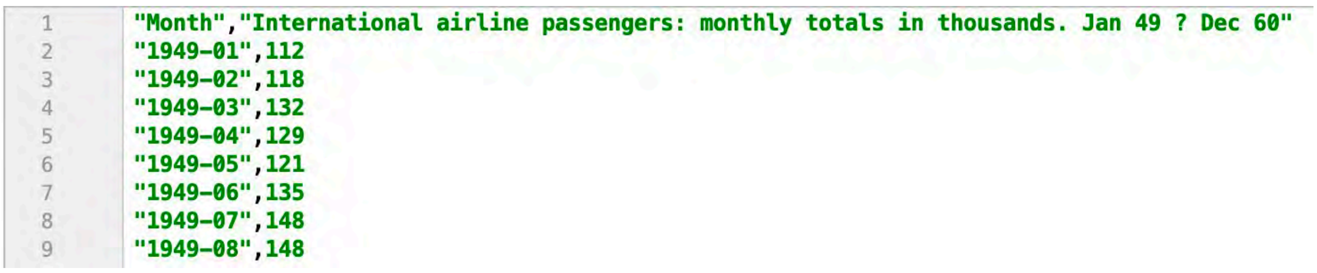
Activity 1 – Time Series Prediction with MLP

In this activity, we will learn:

- ❑ Multilayer Perceptron Regression
- ❑ Multilayer Perceptron Regression Using the Window Method

1) Problem Description: Time Series Prediction

The problem we are going to look at in this lesson is the international airline passengers prediction problem. This is a problem where given a year and a month, the task is to predict the number of international airline passengers in units of 1,000. The data ranges from January 1949 to December 1960 or 12 years, with 144 observations.



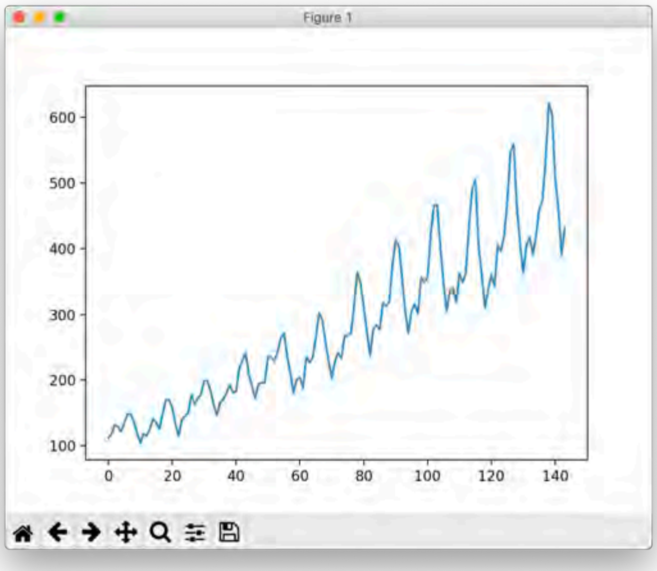
```
"Month","International airline passengers: monthly totals in thousands. Jan 49 ? Dec 60"
"1949-01",112
"1949-02",118
"1949-03",132
"1949-04",129
"1949-05",121
"1949-06",135
"1949-07",148
"1949-08",148
```

We can load this dataset easily using the Pandas library. We are not interested in the date, given that each observation is separated by the same interval of one month. Therefore when we load the dataset we can exclude the first column. The downloaded dataset also has footer information that we can exclude with the skipfooter argument to pandas.read csv() set to 3 for the 3 footer lines.

```
from pandas import read_csv
import matplotlib.pyplot as plt
dataset = read_csv('dataset/international-airline-passengers.csv', usecols=[1], engine='python',
    skipfooter=3)
plt.plot(dataset)
plt.show()
```

** if you see an empty plot, check your /.matplotlib/matplotlibrc and add "backend: MacOSX" for macOS.*

You can see an upward trend in the plot. You can also see some periodicity to the dataset that probably corresponds to the northern hemisphere summer holiday period.



2) Multilayer Perceptron Regression

We can phrase the time series prediction problem as a regression problem. I.e. given the number of passengers (in units of thousands) this month, what is the number of passengers next month. We can write a simple function to convert our single column of data into a two-column dataset. The first column containing this month's (t) passenger count and the second column containing next month's (t+1) passenger count, to be predicted.

a) Use the following code to import the necessary functions and classes we intend to use. Then extract the NumPy array from the dataframe and convert the integer values to floating point values which are more suitable for modelling with a neural network.

```
import numpy
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = read_csv('dataset/international-airline-passengers.csv', usecols=[1],
engine='python',skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype('float32')
```

b) After we model our data and estimate the skill of our model on the training dataset, we need to get an idea of the skill of the model on new unseen data. For a normal classification or regression problem we would do this using k-fold cross-validation. With time series data, the sequence of values is important. A simple method that we can use is to split the ordered dataset into train and test datasets. The code below calculates the index of the split point and separates the data into the training datasets with 67% of the observations that we can use to train our model, leaving the remaining 33% for testing the model.

```
# 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))
```

c) Now we can define a function to create a new dataset as described above. The function takes two arguments, the dataset which is a NumPy array that we want to convert into a dataset and the **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. This default will create a dataset where X is the number of passengers at a given time (t) and Y is the number of passengers at the next time (t+1). It can be configured and we will look at constructing a differently shaped dataset in the next section.

```
# 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)
```

d) Let's take a look at the effect of this function on the first few rows of the dataset.

```
X   Y
112 118
118 132
132 129
129 121
121 135
```

e) Let's use this function to prepare the train and test datasets ready 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)
```

f) We can now fit a Multilayer Perceptron model to the training data. **We use a simple network with 1 input, 1 hidden layer with 8 neurons and an output layer**. The model is fit using mean squared error, if we take the square root gives us an error score in the units of the dataset. We will try a few rough parameters as shown in the configuration below, but by no means is the network listed optimized.

```
# 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, epochs=200, batch_size=2, verbose=2)
```

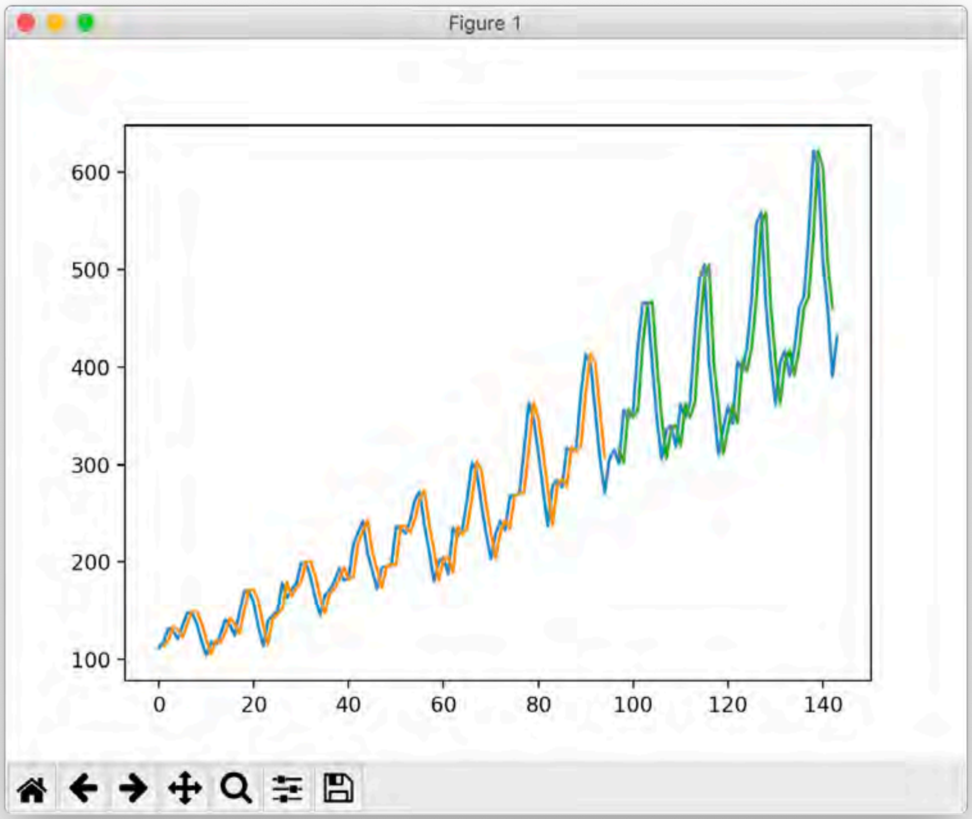
g) Once the model is fit, we can estimate the performance of the model on the train and test datasets. This will give us a point of comparison for new models.

```python
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: %.2f MSE (%.2f RMSE)' % (trainScore, math.sqrt(trainScore)))
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: %.2f MSE (%.2f RMSE)' % (testScore, math.sqrt(testScore)))
```

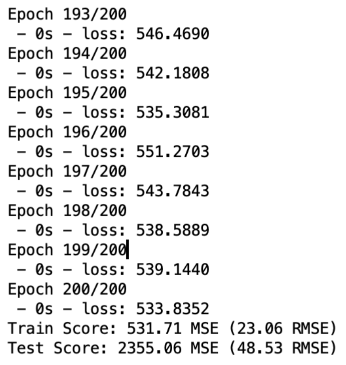
h) Finally, we can generate predictions using the model for both the train and test dataset to get a visual indication of the skill of the model. Because of how the dataset was prepared, we must shift the predictions so that they align on the x-axis with the original dataset. Once prepared, the data is plotted, showing the original dataset in blue, the predictions for the train dataset in orange, the predictions on the unseen test dataset in green.

```python
# 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()
```

Number of Passengers Predicted Using a Simple Multilayer Perceptron Model. Blue=Whole Dataset, Orange=Training, Green=Predictions.



i) Running the complete script, we see that the model has an average error of 23 passengers (in thousands) on the training dataset and 48 passengers (in thousands) on the test dataset.



```
Epoch 193/200
 - 0s - loss: 546.4690
Epoch 194/200
 - 0s - loss: 542.1808
Epoch 195/200
 - 0s - loss: 535.3081
Epoch 196/200
 - 0s - loss: 551.2703
Epoch 197/200
 - 0s - loss: 543.7843
Epoch 198/200
 - 0s - loss: 538.5889
Epoch 199/200
 - 0s - loss: 539.1440
Epoch 200/200
 - 0s - loss: 533.8352
Train Score: 531.71 MSE (23.06 RMSE)
Test Score: 2355.06 MSE (48.53 RMSE)
```

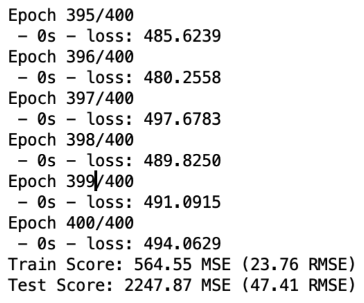
3) Multilayer Perceptron Regression Using the Window Method

We can also phrase the problem so that multiple recent time steps can be used to make the prediction for the next time step. This is called the **window method**, and the size of the window is a parameter that can be tuned for each problem. For example, given the current time (t) we want to predict the value at the next time in the sequence (t+1), we can use the current time (t) as well as the two prior times (t-1 and t-2). When phrased as a regression problem the input variables are t-2, t-1, t and the output variable is t+1.

a) The create dataset() function we wrote in the previous section allows us to create this formulation of the time series problem by increasing the look back argument from 1 to 3. A sample of the dataset with this formulation looks as follows:

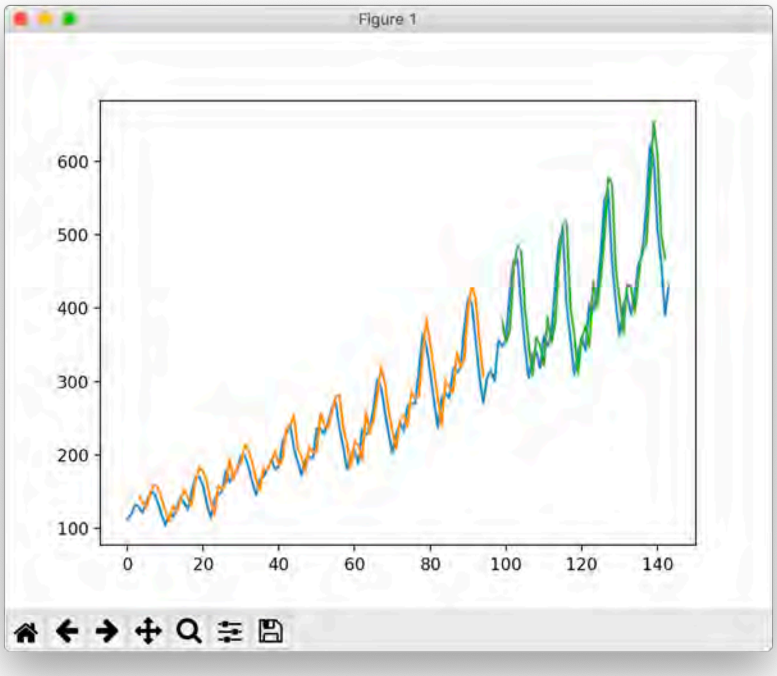
```
X1  X2  X3  Y
112 118 132 129
118 132 129 121
132 129 121 135
129 121 135 148
121 135 148 148
```

b) We can re-run the exercise in the previous section with the larger window size. We will increase the network capacity to handle the additional information. ***The first hidden layer is increased to 12 neurons and a second hidden layer is added with 8 neurons. The number of epochs is also increased to 400. (do this as an exercise)***



```
Epoch 395/400
 - 0s - loss: 485.6239
Epoch 396/400
 - 0s - loss: 480.2558
Epoch 397/400
 - 0s - loss: 497.6783
Epoch 398/400
 - 0s - loss: 489.8250
Epoch 399/400
 - 0s - loss: 491.0915
Epoch 400/400
 - 0s - loss: 494.0629
Train Score: 564.55 MSE (23.76 RMSE)
Test Score: 2247.87 MSE (47.41 RMSE)
```

We can see that the error was reduced compared to that of the previous exercise. Again, the window size and the network architecture were not tuned, this is just a demonstration of how to frame a prediction problem. Taking the square root of the performance scores we can see the average error on the training dataset was 23 passengers (in thousands per month) and the average error on the unseen test set was 47 passengers (in thousands per month).



Prediction of the Number of Passengers using a Simple Multilayer Perceptron Model With Time Lag.
Blue=Whole Dataset, Orange=Training, Green=Predictions

<u>Activity wrap-up:</u>
We learn
- ❏ Multilayer Perceptron Regression
- ❏ Multilayer Perceptron Regression Using the Window Method

Activity 2 – Time Series Prediction with LSTM

In this activity, we will:

❑ LSTM Network for Regression Using The Window Method
❑ LSTM Network for Regression with Time Steps

1. Problem Description:

Like the previous activity, we are going to look at the same international airline passengers prediction problem, but instead of using a Multilayer Perceptron, we will be using LSTM to solve the problem.

2. LSTM Network for Regression

a) 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. We can easily normalize the dataset using the **MinMaxScaler** preprocessing class from the scikit-learn library.

```python
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
```

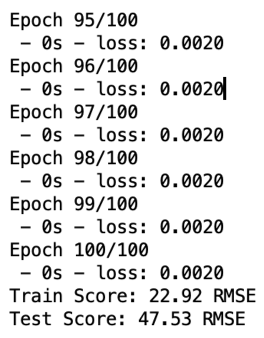
b) The LSTM network expects the input data (X) to be provided with a specific array structure in the form of: **[samples, time steps, features]**. Our prepared data is in the form: [samples, features] and we are framing the problem as one time step for each sample. We can transform the prepared train and test input data into the expected structure using numpy.reshape() as follows

```python
# 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]))
```

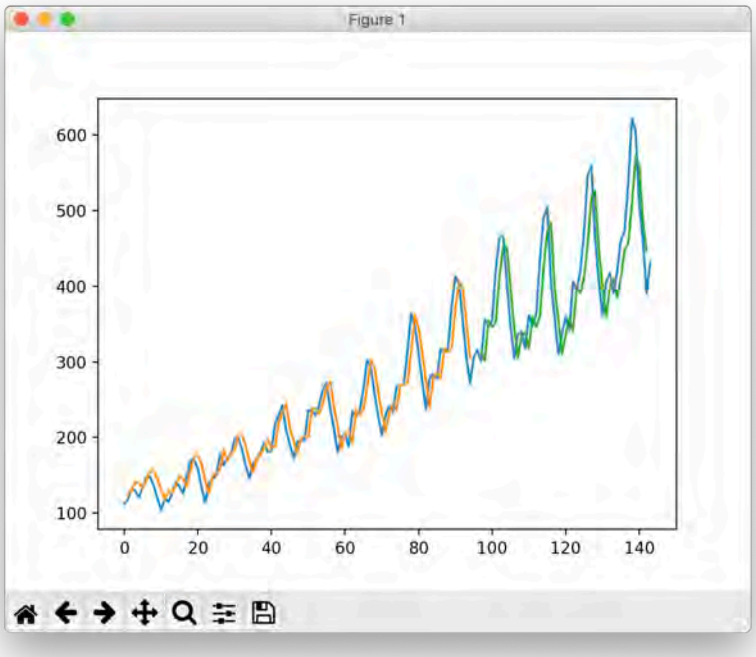
c) We are now ready to design and fit our LSTM network for this problem. The network has a visible layer with 1 input, a hidden layer with 4 LSTM blocks or neurons and an output layer that makes a single value prediction. The default sigmoid activation function is used for the LSTM memory blocks. The network is trained for 100 epochs and a batch size of 1 is used.

```python
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
```

d) Modify the code from the previous activity, run and check your result. We can see that the model did an OK job of fitting both the training and the test datasets. We can see that the model has an average error of about 23 passengers (in thousands) on the training dataset and about 47 passengers (in thousands) on the test dataset.



```
Epoch 95/100
 - 0s - loss: 0.0020
Epoch 96/100
 - 0s - loss: 0.0020
Epoch 97/100
 - 0s - loss: 0.0020
Epoch 98/100
 - 0s - loss: 0.0020
Epoch 99/100
 - 0s - loss: 0.0020
Epoch 100/100
 - 0s - loss: 0.0020
Train Score: 22.92 RMSE
Test Score: 47.53 RMSE
```



3. LSTM Network for Regression Using The Window Method

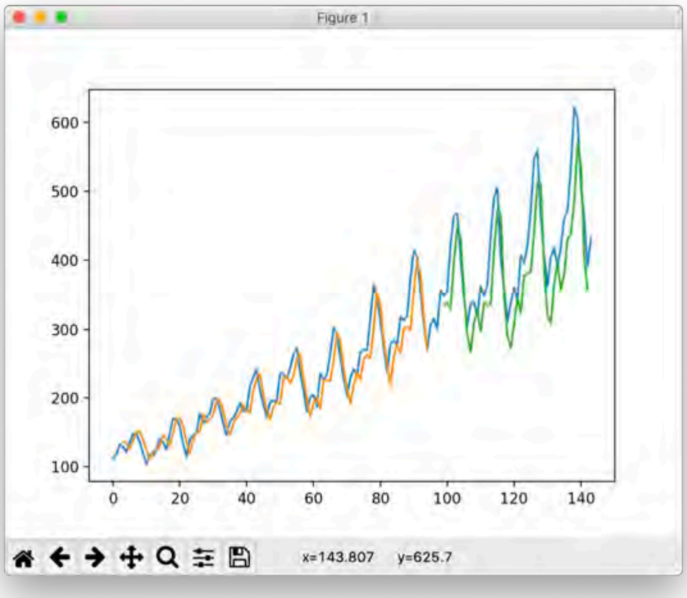
a) We can also phrase the problem so that multiple recent time steps can be used to make the prediction for the next time step. The create dataset() function we created in the previous section allows us to create this formulation of the time series problem by increasing the look back argument from 1 to 3. A sample of the dataset with this formulation looks as follows:

```
X1  X2  X3  Y
112 118 132 129
118 132 129 121
132 129 121 135
129 121 135 148
121 135 148 148
```

b) We can re-run the example in the previous section with the larger window size. Change look_back to 3.

```
Epoch 96/100
 - 0s - loss: 0.0021
Epoch 97/100
 - 0s - loss: 0.0021
Epoch 98/100
 - 0s - loss: 0.0021
Epoch 99/100
 - 0s - loss: 0.0022
Epoch 100/100
 - 0s - loss: 0.0020
Train Score: 24.19 RMSE
Test Score: 58.04 RMSE
```

We can see that the error was increased slightly compared to that of the previous section. The window size and the network architecture were not tuned, this is just a demonstration of how to frame a prediction problem.



4. LSTM Network for Regression with Time Steps

You may have noticed that the data preparation for the LSTM network includes time steps. Some sequence problems may have a varied number of time steps per sample. For example, you may have measurements of a physical machine leading up to a point of failure or a point of surge. Each incident would be a sample, the observations that lead up to the event would be the time steps and the variables observed would be the features.

Time steps provides another way to phrase our time series problem. Like above in the window example, we can take prior time steps in our time series as inputs to predict the output at the next time step. Instead of phrasing the past observations as separate input features, we can use them as time steps of the one input feature, which is indeed a more accurate framing of the problem.

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. For example:

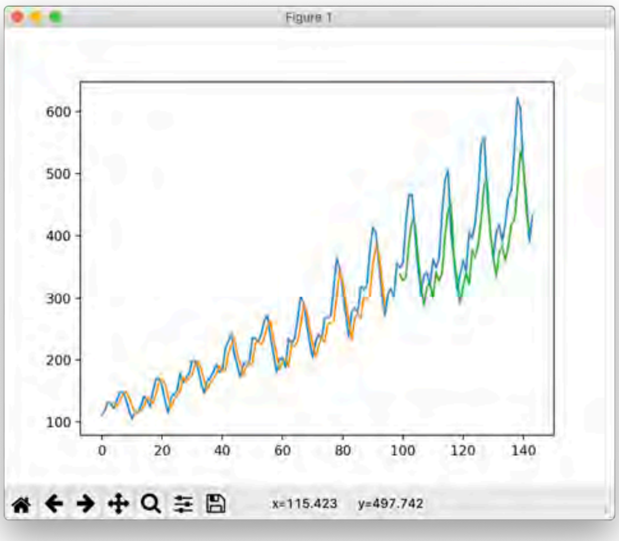
```
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))
```

You will also need to change your LSTM network to reflect this change in training dataset change.

```
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_shape=(look_back,1)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
```

a) Make the modifications and run the script. We can see that the results are slightly better than previous example, and the structure of the input data makes a lot more sense.

```
Epoch 96/100
 - 0s - loss: 0.0021
Epoch 97/100
 - 0s - loss: 0.0021
Epoch 98/100
 - 0s - loss: 0.0020
Epoch 99/100
 - 0s - loss: 0.0021
Epoch 100/100
 - 0s - loss: 0.0020
Train Score: 23.70 RMSE
Test Score: 58.89 RMSE
```



5. LSTM Network for Memory Between Batches
The LSTM network has memory which is capable of remembering across long sequences. ***Normally, the state within the network is reset after each training batch when fitting the model, as well as each call to model.predict() or model.evaluate()***. 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.

a) It requires that the training data not be shuffled 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(), for example:

```
for i in range(100):
  model.fit(trainX, trainY, epochs=1, batch_size=batch_size, verbose=2, shuffle=False)
  model.reset_states()
```

b) Finally, when the LSTM layer is constructed, the **stateful** parameter must be set to **True** and instead of specifying the input dimensions, we must hard code the number of samples in a batch, number of time steps in a sample and number of features in a time step by setting the **batch_input_shape** parameter. For example:

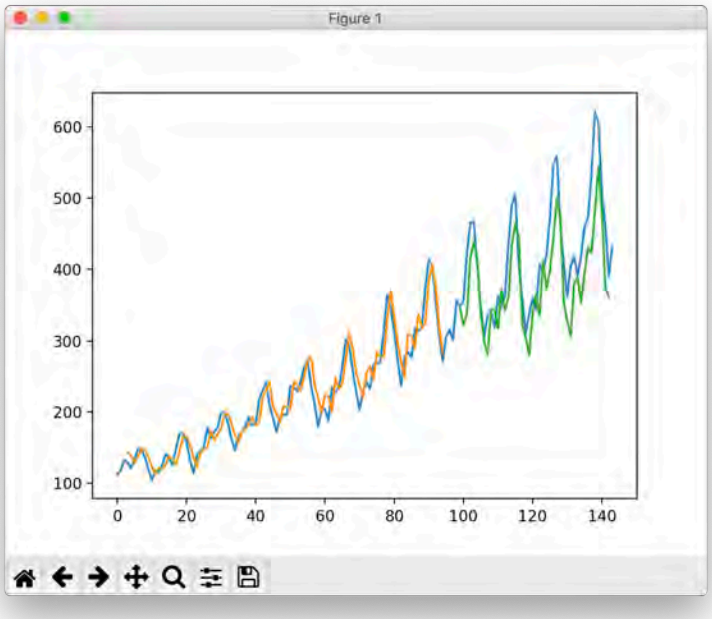
```
model.add(LSTM(4, batch_input_shape=(batch_size, time_steps, features), stateful=True))
```

c) The same batch size must then be used later when evaluating the model and making predictions. For example:

```
model.predict(trainX, batch_size=batch_size)
```

d) Modify the code and run the script. We do see that results are better than some, worse than others. The model may need more modules and may need to be trained for more epochs to internalize the structure of the problem.

```
Epoch 1/1
 - 0s - loss: 0.0017
Epoch 1/1
 - 0s - loss: 0.0017
Epoch 1/1
 - 0s - loss: 0.0017
Epoch 1/1
 - 0s - loss: 0.0017
Epoch 1/1
 - 0s - loss: 0.0017
Epoch 1/1
 - 0s - loss: 0.0017
Train Score: 20.80 RMSE
Test Score: 55.29 RMSE
```



6. Stacked LSTMs Network for Memory Between Batches

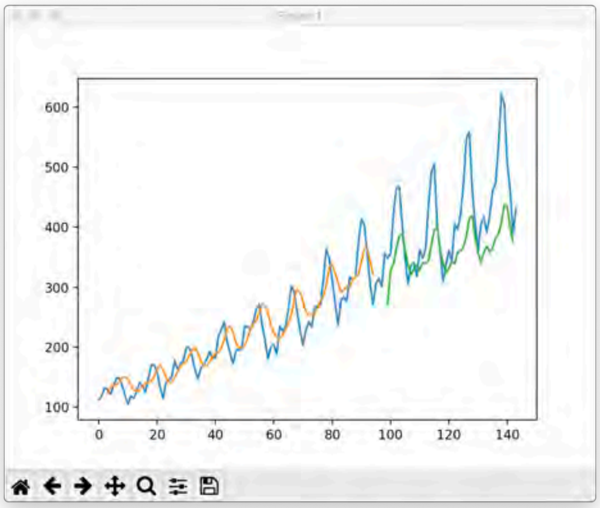
Finally, we will take a look at one of the big benefits of LSTMs, the fact that they can be successfully trained when stacked into deep network architectures. LSTM networks can be stacked in Keras in the same way that other layer types can be stacked.

One addition to the configuration that 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. We can extend the stateful LSTM in the previous section to have two layers, as follows:

```
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))
```

Make the modification and run the script. The predictions on the test dataset are again worse. This is more evidence to suggest the need for additional training epochs.

```
Epoch 1/1
 - 0s - loss: 0.0039
Epoch 1/1
 - 0s - loss: 0.0039
Epoch 1/1
 - 0s - loss: 0.0038
Epoch 1/1
 - 0s - loss: 0.0038
Epoch 1/1
 - 0s - loss: 0.0037
Epoch 1/1
 - 0s - loss: 0.0036
Train Score: 29.81 RMSE
Test Score: 79.50 RMSE
```



<u>Activity wrap-up:</u>

We learn:

❑ LSTM Network for Regression Using The Window Method

❑ LSTM Network for Regression with Time Steps

Activity 3 : Sequence Classification of Movie Reviews

In this activity, we will:

- ❑ A simple LSTM for Sequence Classification
- ❑ LSTM for Sequence Classification With Dropout
- ❑ LSTM and CNN for Sequence Classification

1. Problem Description:

a) The problem that we will use to demonstrate sequence learning in this tutorial is the IMDB movie review sentiment classification problem. We can quickly develop a small LSTM for the IMDB problem and achieve good accuracy. Let's start off by importing the classes and functions required for this model and initializing the random number generator to a constant value to ensure we can easily reproduce the results.

b) The dataset used in this project is the Large Movie Review Dataset often referred to as the IMDB dataset. The IMDB dataset contains 50,000 highly popular movie reviews (good or bad) for training and the same amount again for testing. The data was collected by Stanford researchers and was used in a 2011 paper where a split of 50-50 of the data was used for training and test. An accuracy of 88.89% was achieved.

c) The imdb.load_data() function allows you to load the dataset in a format that is ready for use in neural network and deep learning models. *The words have been replaced by integers that indicate the absolute popularity of the word in the dataset*. The sentences in each review are therefore comprised of a sequence of integers.

d) Calling imdb.load_data() the first time will download the IMDB dataset to your computer and store it in your home directory under ~/.keras/datasets/imdb.pkl as a 32 megabyte file. Usefully, the imdb.load_data() function provides additional arguments including the number of top words to load (where words with a lower integer are marked as zero in the returned data), the number of top words to skip (to avoid the the's) and the maximum length of reviews to support. Let's load the dataset and calculate some properties of it. We will start off by loading some libraries and loading the entire IMDB dataset as a training dataset.

```
import numpy
from keras.datasets import imdb
from matplotlib import pyplot
# load the dataset
(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)
```

e) We can display the shape of the training dataset.

```
# summarize size
print("Training data: ")
print(X.shape)
print(y.shape)
```

f) We can also print the unique class values.

```
# Summarize number of classes
print("Classes: ")
print(numpy.unique(y))
```

2. A simple LSTM for Sequence Classification

a) Let's start off by importing the classes and functions required for this model and initializing the random number generator to a constant value to ensure we can easily reproduce the results.

```
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)
```

b) We need to load the IMDB dataset. We are constraining the dataset to the top 5,000 words. We also split the dataset into train (50%) and test (50%) sets.

```
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
```

c) Next, we need to truncate and pad the input sequences so that they are all the same length for modelling. The model will learn the zero values carry no information so indeed the sequences are not the same length in terms of content, but same length vectors is required to perform the computation in Keras..

```
# 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)
```

d) We can now define, compile and fit our LSTM model. The first layer is the Embedded layer * that uses 32 length vectors to represent each word. The next layer is the LSTM layer with 100 memory units (smart neurons). Finally, because this is a classification problem we use a Dense output layer with a single neuron and a sigmoid activation function to make 0 or 1 predictions for the two classes (good and bad) in the problem. Because it is a binary classification problem, log loss is used as the loss function (binary crossentropy in Keras). The efficient ADAM optimization algorithm is used. The model is fit for only 3 epochs because it quickly overfits the problem. A large batch size of 64 reviews is used to space out weight updates.

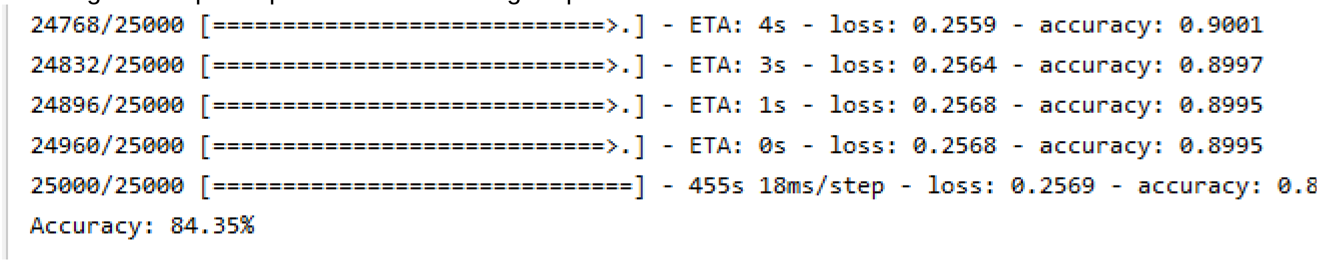
- A breakthrough in the field of natural language processing is called word embedding. This is a technique where words are encoded as real-valued vectors in a high dimensional space, where the similarity between words in terms of meaning translates to closeness in the vector space. Discrete words are mapped to vectors of continuous numbers. This is useful when working with natural language problems with neural networks as we require numbers as input values. Keras provides a convenient way to convert positive integer representations of words into a word embedding by an Embedding layer. The layer takes arguments that define the mapping including the maximum number of expected words also called the vocabulary size (e.g. the largest integer value that will be seen as an input). The layer also allows you to specify the dimensionality for each word vector, called the output dimension. We would like to use a word embedding representation for the IMDB dataset. Let us say that we are only interested in the first 5,000 most used words in the dataset. Therefore, our vocabulary size will be 5,000. We can choose to use a 32-dimensional vector to represent each word. Finally, we may choose to cap the maximum review length at 500 words, truncating reviews longer than that and padding reviews shorter than that with 0 values.

```
# create the model
embedding_vecor_length = 32
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)
```

e) Once fit, we estimate the performance of the model on unseen reviews.

```
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

f) Running the script will produce the following output.



```
24768/25000 [============================>.] - ETA: 4s - loss: 0.2559 - accuracy: 0.9001
24832/25000 [============================>.] - ETA: 3s - loss: 0.2564 - accuracy: 0.8997
24896/25000 [============================>.] - ETA: 1s - loss: 0.2568 - accuracy: 0.8995
24960/25000 [============================>.] - ETA: 0s - loss: 0.2568 - accuracy: 0.8995
25000/25000 [==============================] - 455s 18ms/step - loss: 0.2569 - accuracy: 0.8
Accuracy: 84.35%
```

You can see that this simple LSTM with little tuning achieves near state-of-the-art results on the IMDB problem. Importantly, this is a template that you can use to apply LSTM networks to your own sequence classification problems.

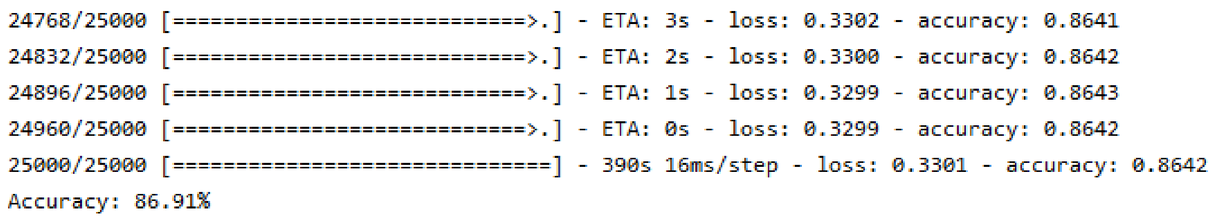
3. LSTM for Sequence Classification With Dropout

Recurrent Neural networks like LSTM generally have the problem of overfitting. Dropout can be applied between layers using the Dropout Keras layer.

a) We can do this easily by adding new Dropout layers between the Embedding and LSTM layers and the LSTM and Dense output layers. For example:

```
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(Dropout(0.2))
model.add(LSTM(100))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
```

b) Modify your code and run it.



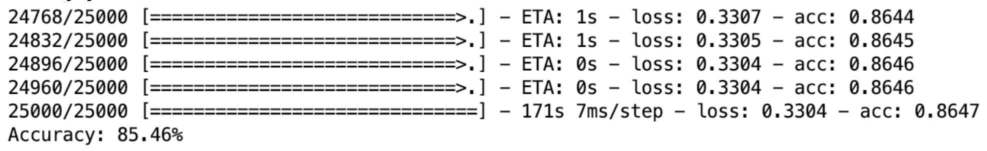
```
24768/25000 [============================>.] - ETA: 3s - loss: 0.3302 - accuracy: 0.8641
24832/25000 [============================>.] - ETA: 2s - loss: 0.3300 - accuracy: 0.8642
24896/25000 [============================>.] - ETA: 1s - loss: 0.3299 - accuracy: 0.8643
24960/25000 [============================>.] - ETA: 0s - loss: 0.3299 - accuracy: 0.8642
25000/25000 [==============================] - 390s 16ms/step - loss: 0.3301 - accuracy: 0.8642
Accuracy: 86.91%
```

We can see dropout having the desired impact on training with a slightly slower trend in convergence and in this case a lower final accuracy. The model could probably use a few more epochs of training and may achieve a higher skill (try it and see)

c) 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 for configuring the input dropout and recurrent dropout for configuring the recurrent dropout. For example, we can modify the first example to add dropout to the input and recurrent connections as follows:

```
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length,
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
```

d) Modify your code and run it.



```
24768/25000 [============================>.] - ETA: 1s - loss: 0.3307 - acc: 0.8644
24832/25000 [============================>.] - ETA: 1s - loss: 0.3305 - acc: 0.8645
24896/25000 [============================>.] - ETA: 0s - loss: 0.3304 - acc: 0.8646
24960/25000 [============================>.] - ETA: 0s - loss: 0.3304 - acc: 0.8646
25000/25000 [==============================] - 171s 7ms/step - loss: 0.3304 - acc: 0.8647
Accuracy: 85.46%
```

We can see that the LSTM specific dropout has a more pronounced effect 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. Dropout is a powerful technique for combating overfitting in your LSTM models and it is a good idea to try both methods, but you may bet better results with the gate-specific dropout provided in Keras.

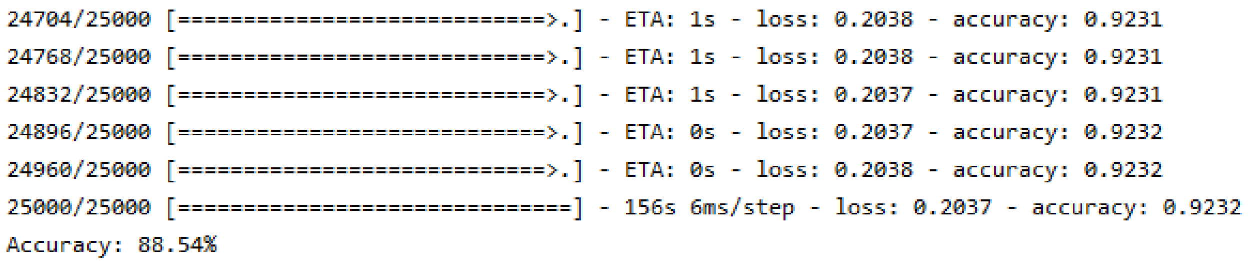
4. LSTM and CNN for Sequence Classification

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.

a) 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. We can use a smallish set of 32 features with a small filter length of 3. The pooling layer can use the standard length of 2 to halve the feature map size. For example, we would create the model as follows:

```
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(Conv1D(filters=32, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
```

b) Modify your code and run it.



```
24704/25000 [============================>.] - ETA: 1s - loss: 0.2038 - accuracy: 0.9231
24768/25000 [============================>.] - ETA: 1s - loss: 0.2038 - accuracy: 0.9231
24832/25000 [============================>.] - ETA: 1s - loss: 0.2037 - accuracy: 0.9231
24896/25000 [============================>.] - ETA: 0s - loss: 0.2037 - accuracy: 0.9232
24960/25000 [============================>.] - ETA: 0s - loss: 0.2038 - accuracy: 0.9232
25000/25000 [==============================] - 156s 6ms/step - loss: 0.2037 - accuracy: 0.9232
Accuracy: 88.54%
```

We can see that we achieve similar results to the first example although with less weights and faster training time. We can expect that even better results could be achieved if this example was further extended to use dropout.

<u>Activity wrap-up:</u>
We learn:
- ❑ A simple LSTM for Sequence Classification
- ❑ LSTM for Sequence Classification With Dropout
- ❑ LSTM and CNN for Sequence Classification

Activity 4 – Develop a 1D GAN from scratch

In this activity, we will:

- ❑ Select a One Dimensional Function
- ❑ Define a Discriminator Model
- ❑ Define a Generator Model
- ❑ Training the Generator Model
- ❑ Evaluating the Performance of the GAN
- ❑ Complete Example of Training the GAN

1. Select a One Dimensional Function:

The first step is to select a one-dimensional function to model. Something of the form: ***y=f(x).***

a) Where x are input values and y are the output values of the function. Specifically, we want a function that we can easily understand and plot. This will help in both setting an expectation of what the model should be generating and in using a visual inspection of generated examples to get an idea of their quality.

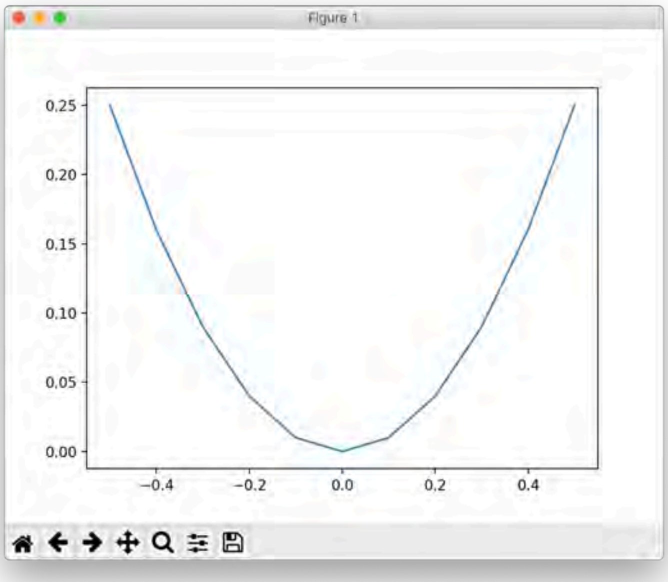
b) We will use a simple function of x^2; that is, the function will return the square of the input. We can define the function in Python as follows:

```python
# simple function
def calculate(x):
  return x * x
```

c) We can define the input domain as real values between -0.5 and 0.5 and calculate the output value for each input value in this linear range, then plot the results to get an idea of how inputs relate to outputs. The complete example is listed below.

```python
# demonstrate simple x^2 function
from matplotlib import pyplot

# simple function
def calculate(x):
  return x * x

# define inputs
inputs = [-0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5]

# calculate outputs
outputs = [calculate(x) for x in inputs]

# plot the result
pyplot.plot(inputs, outputs)
pyplot.show()
```

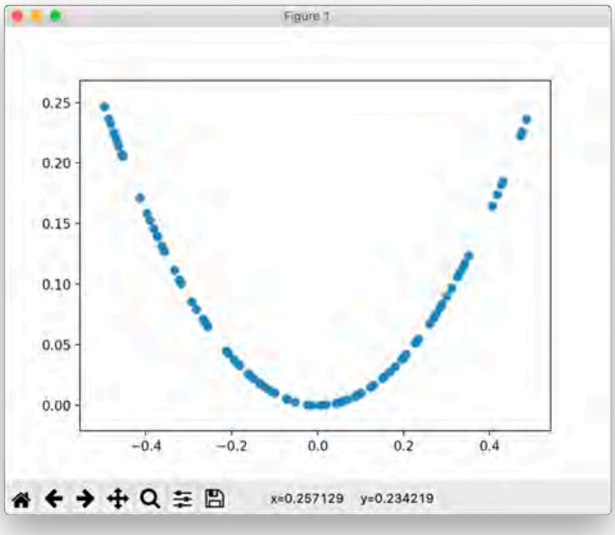
d) Running the example calculates the output value for each input value and creates a plot of input vs. output values. We can see that values far from 0.0 result in larger output values, whereas values close to zero result in smaller output values, and that this behaviour is symmetrical around zero.



e) We can generate random samples or points from the function. This can be achieved by generating random values between -0.5 and 0.5 and calculating the associated output value. Repeating this many times will give a sample of points from the function, e.g. real samples. Plotting these samples using a scatter plot will show the same u-shape plot, although comprised of the individual random samples. The complete example is listed below.

```
# example of generating random samples from X^2
from numpy.random import rand
from numpy import hstack
from matplotlib import pyplot

# generate randoms sample from x^2
def generate_samples(n=100):
  # generate random inputs in [-0.5, 0.5]
  X1 = rand(n) - 0.5
  # generate outputs X^2 (quadratic)
  X2 = X1 * X1
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  return hstack((X1, X2))

# generate samples
data = generate_samples()
# plot samples
pyplot.scatter(data[:, 0], data[:, 1])
pyplot.show()
```

f) Running the example generates 100 random inputs and their calculated output and plots the sample as a scatter plot, showing the familiar u-shape.



g) We can use this function as a starting point for generating real samples for our discriminator function. Specifically, ***a sample is comprised of a vector with two elements, one for the input and one for the output of our one-dimensional function***. We can also imagine how a generator model could generate new samples that we can plot and compare to the expected u-shape of the X^2 function. Specifically, a generator would output a vector with two elements: one for the input and one for the output of our one-dimensional function.

2. Define a Discriminator Model

a) The model must take a sample from our problem, such as a vector with two elements, and output a classification prediction as to whether the sample is real or fake.
- **Inputs**: Sample with two real values.
- **Outputs**: Binary classification, likelihood the sample is real (or fake).

b) The problem is very simple, meaning that we don't need a complex neural network to model it. The discriminator model will have one hidden layer with 25 nodes and we will use the ***ReLU*** activation function and an appropriate weight initialization method called ***He*** weight initialization. The output layer will have one node for the ***binary*** classification using the ***sigmoid*** activation function.
The model will minimize the binary cross-entropy loss function, and the Adam version of stochastic gradient descent will be used because it is very effective.
The define_discriminator() function below defines and returns the discriminator model.

The function parameterizes the number of inputs to expect, which defaults to two.

```
# define the standalone discriminator model
def define_discriminator(n_inputs=2):
  model = Sequential()
  model.add(Dense(25, activation="relu", kernel_initializer="he_uniform", input_dim=n_inputs))
  model.add(Dense(1, activation="sigmoid"))
  # compile model
  model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
  return model
```

c) Add the following code to create the discriminator model and summarizes it.

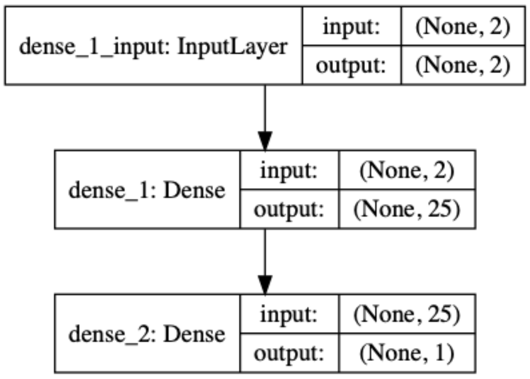
```
model = define_discriminator()
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file="discriminator_plot.png", show_shapes=True, show_layer_names=True)
```

d) The complete code.

```
# define the discriminator model
from keras.models import Sequential
from keras.layers import Dense
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(n_inputs=2):
  model = Sequential()
  model.add(Dense(25, activation="relu", kernel_initializer="he_uniform", input_dim=n_inputs))
  model.add(Dense(1, activation="sigmoid"))
  # compile model
  model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
  return model

# define the discriminator model
model = define_discriminator()

# summarize the model
model.summary()

# plot the model
plot_model(model, to_file="discriminator_plot.png", show_shapes=True, show_layer_names=True)
```

Running the example defines the discriminator model and summarizes it.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 25)                75
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 26
=================================================================
Total params: 101
Trainable params: 101
Non-trainable params: 0
_________________________________________________________________
```



e) We could start training this model now with real examples with a class label of one and randomly generated samples with a class label of zero. There is no need to do this, but the elements we will develop will be useful later, and it helps to see that the discriminator is just a normal neural network model. First, we can update our **generate_samples()** function from the prediction section and call it **generate_real_samples()** and have it also return the output class labels for the real samples, specifically, an array of 1 values, where class = 1 means real.

```
# generate n real samples with class labels
def generate_real_samples(n):
  # generate inputs in [-0.5, 0.5]
  X1 = rand(n) - 0.5
  # generate outputs X^2
  X2 = X1 * X1
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # generate class labels
  y = ones((n, 1))
  return X, y
```

f) Next, we can create a copy of this function for creating fake examples. In this case, we will generate random values in the range -1 and 1 for both elements of a sample. The output class label for all of these examples is 0. This function will act as our fake generator model

```
# generate n fake samples with class labels
def generate_fake_samples(n):
  # generate inputs in [-1, 1]
  X1 = -1 + rand(n) * 2
  # generate outputs in [-1, 1]
  X2 = -1 + rand(n) * 2
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # generate class labels
  y = zeros((n, 1))
  return X, y
```

g) Next, we need a function to train and evaluate the discriminator model. This can be achieved by manually enumerating the training epochs and for each epoch generating a half batch of real examples and a half batch of fake examples, and updating the model on each, e.g. one whole batch of examples. The train() function could be used, but in this case, we will use the **train_on_batch()** function directly. The model can then be evaluated on the generated examples and we can report the classification accuracy on the real and fake samples. The **train_discriminator()** function below implements this, training the model for 1,000 batches and using 128 samples per batch (64 fake and 64 real).

```
# train the discriminator model
def train_discriminator(model, n_epochs=1000, n_batch=128):
  half_batch = int(n_batch / 2)
  # run epochs manually
  for i in range(n_epochs):
    # generate real examples
    X_real, y_real = generate_real_samples(half_batch)
    # update model
    model.train_on_batch(X_real, y_real)
    # generate fake examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update model
    model.train_on_batch(X_fake, y_fake)
    # evaluate the model
    _, acc_real = model.evaluate(X_real, y_real, verbose=0)
    _, acc_fake = model.evaluate(X_fake, y_fake, verbose=0)
    print(i, acc_real, acc_fake)
```

h) The complete example is listed below.

```
# define and fit a discriminator model
from numpy import zeros
from numpy import ones
from numpy import hstack
from numpy.random import rand
from keras.models import Sequential
from keras.layers import Dense

# define the standalone discriminator model
def define_discriminator(n_inputs=2):
  model = Sequential()
  model.add(Dense(25, activation="relu", kernel_initializer="he_uniform", input_dim=n_inputs))
  model.add(Dense(1, activation="sigmoid"))
  # compile model
  model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
  return model

# generate n real samples with class labels
def generate_real_samples(n):
  # generate inputs in [-0.5, 0.5]
  X1 = rand(n) - 0.5
  # generate outputs X^2
  X2 = X1 * X1
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # generate class labels
  y = ones((n, 1))
  return X, y

# generate n fake samples with class labels
def generate_fake_samples(n):
  # generate inputs in [-1, 1]
  X1 = -1 + rand(n) * 2
  # generate outputs in [-1, 1]
  X2 = -1 + rand(n) * 2
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # generate class labels
  y = zeros((n, 1))
  return X, y


# train the discriminator model
def train_discriminator(model, n_epochs=1000, n_batch=128):
   half_batch = int(n_batch / 2)
   # run epochs manually
   for i in range(n_epochs):
        # generate real examples
        X_real, y_real = generate_real_samples(half_batch)
        # update model
        model.train_on_batch(X_real, y_real)
        # generate fake examples
        X_fake, y_fake = generate_fake_samples(half_batch)
        # update model
        model.train_on_batch(X_fake, y_fake)
        # evaluate the model
        _, acc_real = model.evaluate(X_real, y_real, verbose=0)
        _, acc_fake = model.evaluate(X_fake, y_fake, verbose=0)
        print(i, acc_real, acc_fake)


# define the discriminator model
model = define_discriminator()

# fit the model
train_discriminator(model)
```

Running the example generates real and fake examples and updates the model, then evaluates the model on the same examples and prints the classification accuracy.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the model rapidly learns to correctly identify the real examples with perfect accuracy and is very good at identifying the fake examples with 80% to 90% accuracy.

```
995 1.0 0.84375
996 1.0 0.828125
997 1.0 0.90625
998 1.0 0.84375
999 1.0 0.875
```

Training the discriminator model is straightforward. The goal is to train a generator model, not a discriminator model, and that is where the complexity of GANs truly lies.

3. Define a Generator Model

a) The generator model takes as input a point from the latent space and generates a new sample, e.g. a vector with both the input and output elements of our function, e.g. x and x^2. A latent variable is a hidden or unobserved variable, and a latent space is a multi-dimensional vector space of these variables. We can define the size of the latent space for our problem and the shape or distribution of variables in the latent space

b) After training, points in the latent space will correspond to points in the output space, e.g. in the space of generated samples. We will define a small latent space of five dimensions and use the standard approach in the GAN literature of using a Gaussian distribution for each variable in the latent space. **We will generate new**

inputs by drawing random numbers from a standard Gaussian distribution, i.e. mean of zero and a standard deviation of one.

   a. **Inputs**: Point in latent space, e.g. a five-element vector of Gaussian random numbers.
   b. **Outputs**: Two-element vector representing a generated sample for our function (x and x^2).

c) The generator model will be small like the discriminator model. It will have a single ***Dense*** hidden layer with fifteen nodes and will use the ***ReLU*** activation function and ***He*** weight initialization. The output layer will have two nodes for the two elements in a generated vector and will use a linear activation function. A linear activation function is used because we know we want the generator to output a vector of real values and the scale will be [-0.5, 0.5] for the first element and about [0.0, 0.25] for the second element.

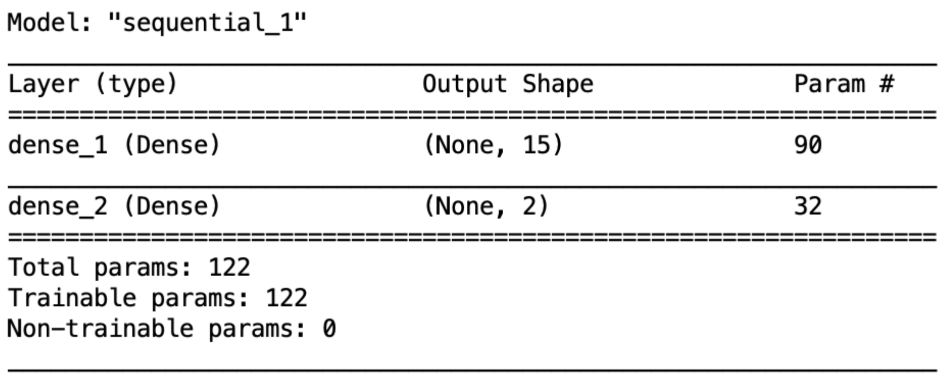
d) **The model is not compiled**. The reason for this is that the generator model is not fit directly. The define_generator() function below defines and returns the generator model. The size of the latent dimension is parameterized in case we want to play with it later, and the output shape of the model is also parameterized, matching the function for defining the discriminator model.

```
# define the standalone generator model
def define_generator(latent_dim, n_outputs=2):
  model = Sequential()
  model.add(Dense(15, activation=”relu”, kernel_initializer=”he_uniform”, input_dim=latent_dim))
  model.add(Dense(n_outputs, activation=”linear”))
  return model
```

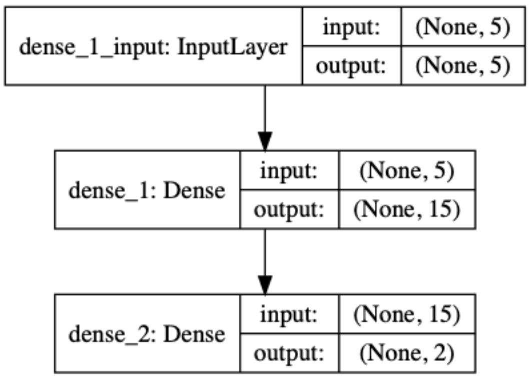
e) The code to define the generator.

```
# define the generator model
from keras.models import Sequential
from keras.layers import Dense
from keras.utils.vis_utils import plot_model

# define the standalone generator model
def define_generator(latent_dim, n_outputs=2):
  model = Sequential()
  model.add(Dense(15, activation="relu", kernel_initializer="he_uniform", input_dim=latent_dim))
  model.add(Dense(n_outputs, activation="linear"))
  return model

# define the discriminator model
model = define_generator(5)

# summarize the model
model.summary()

# plot the model
plot_model(model, to_file="generator_plot.png", show_shapes = True, show_layer_names = True)
```

Running the example defines the generator model and summarizes it.



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 15)                90
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 32
=================================================================
Total params: 122
Trainable params: 122
Non-trainable params: 0
_________________________________________________________________
```



f) This model cannot do much at the moment. Nevertheless, we can demonstrate how to use it to generate samples. This is not needed, but again, some of these elements may be useful later.

The first step is to generate new random points in the latent space. We can achieve this by calling the randn() NumPy function for generating arrays of random numbers drawn from a standard Gaussian. The array of random numbers can then be reshaped into samples: that is n rows with five elements per row. The **generate_latent_points()** function below implements this and generates the desired number of points in the latent space that can be used as input to the generator model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n):
  # generate points in the latent space
  x_input = randn(latent_dim * n)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n, latent_dim)
  return x_input
```

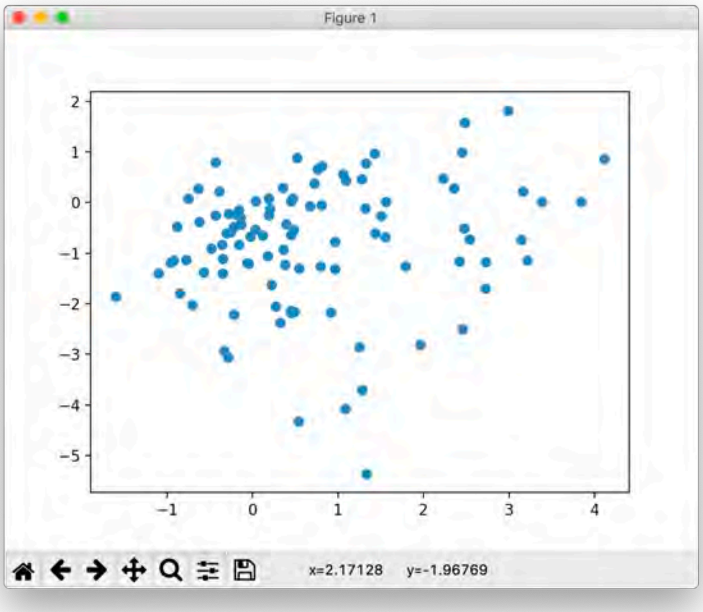
g) Next, we can use the generated points as input the generator model to generate new samples, then plot the samples. The **generate_fake_samples()** function below implements this, where the defined generator and size of the latent space are passed as arguments, along with the number of points for the model to generate.

```
# use the generator to generate n fake examples and plot the results
def generate_fake_samples(generator, latent_dim, n):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n)
  # predict outputs
  X = generator.predict(x_input)
  # plot the results
  pyplot.scatter(X[:, 0], X[:, 1])
  pyplot.show()
```

h) The complete code.

```
# define and use the generator model
from numpy.random import randn
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot

# define the standalone generator model
def define_generator(latent_dim, n_outputs=2):
  model = Sequential()
  model.add(Dense(15, activation="relu", kernel_initializer="he_uniform",
  input_dim=latent_dim))
  model.add(Dense(n_outputs, activation="linear"))
  return model

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n):
  # generate points in the latent space
  x_input = randn(latent_dim * n)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n, latent_dim)
  return x_input


# use the generator to generate n fake examples and plot the results
def generate_fake_samples(generator, latent_dim, n):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n)
  # predict outputs
  X = generator.predict(x_input)
  # plot the results
```

```
    pyplot.scatter(X[:, 0], X[:, 1])
    pyplot.show()

# size of the latent space
latent_dim = 5

# define the discriminator model
model = define_generator(latent_dim)

# generate and plot generated samples
generate_fake_samples(model, latent_dim, 100)
```



Running the example generates 100 random points from the latent space, uses this as input to the generator and generates 100 fake samples from our one-dimensional function domain. As the generator has not been trained, the generated points are complete rubbish, as we expect, but we can imagine that as the model is trained, these points will slowly begin to resemble the target function and its u-shape.

4. Training the Generator Model

a) The weights in the generator model are updated based on the performance of the discriminator model. When the discriminator is good at detecting fake samples, the generator is updated more (via a larger error gradient), and when the discriminator model is relatively poor or confused when detecting fake samples, the generator model is updated less.

b) There may be many ways to implement this using the Keras API, but perhaps the simplest approach is to create a new model that subsumes or encapsulates the generator and discriminator models. Specifically, a new GAN model can be defined that stacks the generator and discriminator such that the generator receives as input random points in the latent space, generates samples that are fed into the discriminator model directly, classified, and the output of this larger model can be used to update the model weights of the generator. To be clear, we are not talking about a new third model, just a logical third model that uses the already-defined layers and weights from the standalone generator and discriminator models.

c) Only the discriminator is concerned with distinguishing between real and fake examples; therefore, the discriminator model can be trained in a standalone manner on examples of each.

d) The generator model is only concerned with the discriminator's performance on fake examples. Therefore, we will mark all of the layers in the discriminator as not trainable when it is part of the GAN model so that they cannot be updated and overtrained on fake examples.

e) When training the generator via this subsumed GAN model, there is one more important change. The generator wants the discriminator to think that the samples output by the generator are real, not fake. Therefore, when the generator is trained as part of the GAN model, we will mark the generated samples as real (class = 1). We can imagine that the discriminator will then classify the generated samples as not real (class = 0) or a low probability of being real (0.3 or 0.5).

f) The backpropagation process used to update the model weights will see this as a large error and will update the model weights (i.e. only the weights in the generator) to correct for this error, in turn making the generator better at generating plausible fake samples.

- **Inputs**: Point in latent space, e.g. a five-element vector of Gaussian random numbers.

- **Outputs**: Binary classification, likelihood the sample is real (or fake).

g) The define_gan() function below takes as arguments the already-defined generator and discriminator models and creates the new logical third model subsuming these two models. The weights in the discriminator are marked as not trainable, which only affects the weights as seen by the GAN model and not the standalone discriminator model. The GAN model then uses the same binary cross-entropy loss function as the discriminator and the efficient Adam version of stochastic gradient descent.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  model.compile(loss=”binary_crossentropy”, optimizer=”adam”)
  return model
```

Note: Making the discriminator not trainable is a clever trick in the Keras API. The trainable property impacts the model when it is compiled. The discriminator model was compiled with trainable layers, therefore the model weights in those layers will be updated when the standalone model is updated via calls to train_on_batch(). The discriminator model was marked as not trainable, added to the GAN model, and compiled. In this model, the model weights of the discriminator model are not trainable and cannot be changed when the GAN model is updated via calls to train_on_batch().

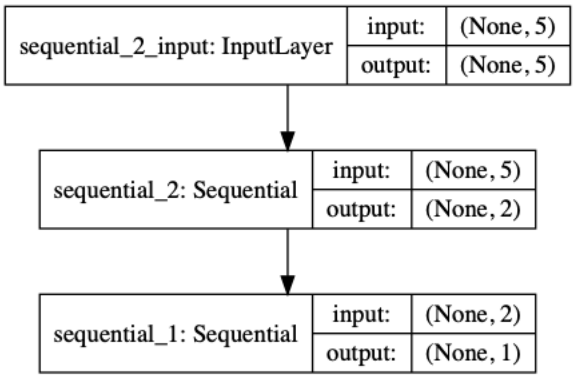
h) The complete example of creating the discriminator, generator, and composite model is listed below.

```
# demonstrate creating the three models in the gan
from keras.models import Sequential
from keras.layers import Dense
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(n_inputs=2):
  model = Sequential()
  model.add(Dense(25, activation="relu", kernel_initializer="he_uniform", input_dim=n_inputs))
  model.add(Dense(1, activation="sigmoid"))
  # compile model
  model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
  return model

# define the standalone generator model
def define_generator(latent_dim, n_outputs=2):
  model = Sequential()
  model.add(Dense(15, activation="relu", kernel_initializer="he_uniform",input_dim=latent_dim))
  model.add(Dense(n_outputs, activation="linear"))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  model.compile(loss="binary_crossentropy", optimizer="adam")
  return model

# size of the latent space
latent_dim = 5

# create the discriminator
discriminator = define_discriminator()

# create the generator
generator = define_generator(latent_dim)

# create the gan
gan_model = define_gan(generator, discriminator)

# summarize gan model
gan_model.summary()
```

```

# plot gan model
plot_model(gan_model, to_file="gan_plot.png", show_shapes=True, show_layer_names=True)
```

Running the example first creates a summary of the composite model. You might get a UserWarning about not calling the compile() function that you can safely ignore.

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_2 (Sequential)    (None, 2)                 122
_________________________________________________________________
sequential_1 (Sequential)    (None, 1)                 101
=================================================================
Total params: 223
Trainable params: 122
Non-trainable params: 101
_________________________________________________________________
```



i) Training the composite model involves generating a batch-worth of points in the latent space via the **generate_latent_points()** function in the previous section, and class = 1 labels and calling the **train_on_batch()** function. The **train_gan()** function below demonstrates this, although it is pretty uninteresting as only the generator will be updated each epoch, leaving the discriminator with default model weights.

```
# train the composite model
def train_gan(gan_model, latent_dim, n_epochs=10000, n_batch=128):
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare points in latent space as input for the generator
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator’s error
    gan_model.train_on_batch(x_gan, y_gan)
```

j) **Instead, what is required is that we first update the discriminator model with real and fake samples, then update the generator via the composite model.** This requires combining elements from the **train_discriminator()** function defined in the discriminator section and the **train_gan()** function defined above. It also requires that the **generate_fake_samples()** function use the generator model to generate fake samples instead of generating random numbers. The complete train function for updating the discriminator model and the generator (via the composite model) is listed below.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, latent_dim, n_epochs=10000, n_batch=128):
  # determine half the size of one batch, for updating the discriminator
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare real samples
    x_real, y_real = generate_real_samples(half_batch)
    # prepare fake examples
    x_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator
    d_model.train_on_batch(x_real, y_real)
    d_model.train_on_batch(x_fake, y_fake)
    # prepare points in latent space as input for the generator
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator’s error
    gan_model.train_on_batch(x_gan, y_gan)
```

5. Evaluating the Performance of the GAN

a) Generally, **there are no objective ways to evaluate the performance of a GAN model**.

b) In this specific case, we can devise an objective measure for the generated samples as we know the true underlying input domain and target function and can calculate an objective error measure. Nevertheless, we will not calculate this objective error score in this tutorial. Instead, we will use the subjective approach used in most GAN applications. Specifically, we will use the generator to generate new samples and inspect them relative to real samples from the domain.

c) First, we can use the generate real samples() function developed in the discriminator part above to generate real examples. Creating a scatter plot of these examples will create the familiar u-shape of our target function.

```
# generate n real samples with class labels
def generate_real_samples(n):
  # generate inputs in [-0.5, 0.5]
  X1 = rand(n) - 0.5
  # generate outputs X^2
  X2 = X1 * X1
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # generate class labels
  y = ones((n, 1))
  return X, y
```

d) Next, we can use the generator model to generate the same number of fake samples. This requires first generating the same number of points in the latent space via the generate_latent_points() function developed in the generator section above. These can then be passed to the generator model and used to generate samples that can also be plotted on the same scatter plot.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n):
  # generate points in the latent space
  x_input = randn(latent_dim * n)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n, latent_dim)
  return x_input
```

e) The generate_fake_samples() function below generates these fake samples and the associated class label of 0 which will be useful later.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n, 1))
  return X, y
```

f) Having both samples plotted on the same graph allows them to be directly compared to see if the same input and output domain are covered and whether the expected shape of the target function has been appropriately captured, at least subjectively. The summarize performance() function below can be called any time during training to create a scatter plot of real and generated points to get an idea of the current capability of the generator model.

```
# plot real and fake points
def summarize_performance(generator, latent_dim, n=100):
  # prepare real samples
  x_real, y_real = generate_real_samples(n)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(generator, latent_dim, n)
  # scatter plot real and fake data points
  pyplot.scatter(x_real[:, 0], x_real[:, 1], color=”red”)
  pyplot.scatter(x_fake[:, 0], x_fake[:, 1], color=”blue”)
  pyplot.show()
```

g) We may also be interested in the performance of the discriminator model at the same time. Specifically, we are interested to know how well the discriminator model can correctly identify real and fake samples. A good generator model should make the discriminator model confused, resulting in a classification accuracy closer to 50% on real and fake examples. We can update the summarize performance() function to also take the discriminator and current epoch number as arguments and report the accuracy on the sample of real and fake examples. It will also generate a plot of synthetic plots and save it to file for later review.

```
# evaluate the discriminator and plot real and fake points
def summarize_performance(epoch, generator, discriminator, latent_dim, n=100):
  # prepare real samples
  x_real, y_real = generate_real_samples(n)
  # evaluate discriminator on real examples
  _, acc_real = discriminator.evaluate(x_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(generator, latent_dim, n)
  # evaluate discriminator on fake examples
  _, acc_fake = discriminator.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print(epoch, acc_real, acc_fake)
  # scatter plot real and fake data points
  pyplot.scatter(x_real[:, 0], x_real[:, 1], color=”red”)
  pyplot.scatter(x_fake[:, 0], x_fake[:, 1], color=”blue”)
  # save plot to file
  filename = “generated_plot_e%03d.png” % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()
```

h) This function can then be called periodically during training. For example, if we choose to train the models for 10,000 iterations, it may be interesting to check-in on the performance of the model every 2,000 iterations. We can achieve this by parameterizing the frequency of the check-in via n eval argument, and calling the **summarize_performance()** function from the train() function after the appropriate number of iterations. The updated version of the train() function with this change is listed below.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, latent_dim, n_epochs=10000, n_batch=128,n_eval=2000):
  # determine half the size of one batch, for updating the discriminator
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare real samples
    x_real, y_real = generate_real_samples(half_batch)
    # prepare fake examples
    x_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator
    d_model.train_on_batch(x_real, y_real)
    d_model.train_on_batch(x_fake, y_fake)
    # prepare points in latent space as input for the generator
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator”s error
    gan_model.train_on_batch(x_gan, y_gan)
    # evaluate the model every n_eval epochs
    if (i+1) % n_eval == 0:
      summarize_performance(i, g_model, d_model, latent_dim)
```

6. Complete Example of Training the GAN

a) We now have everything we need to train and evaluate a GAN on our chosen one-dimensional function. The complete example is listed below.

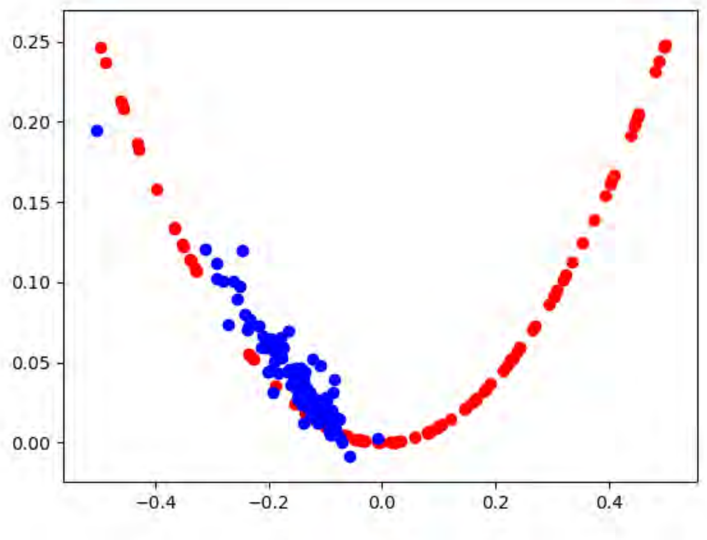
```
# train a generative adversarial network on a one-dimensional function
from numpy import hstack
from numpy import zeros
from numpy import ones
from numpy.random import rand
from numpy.random import randn
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(n_inputs=2):
  model = Sequential()
  model.add(Dense(25, activation="relu", kernel_initializer="he_uniform",input_dim=n_inputs))
  model.add(Dense(1, activation="sigmoid"))
  # compile model
  model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
  return model

# define the standalone generator model
def define_generator(latent_dim, n_outputs=2):
  model = Sequential()
  model.add(Dense(15, activation="relu", kernel_initializer="he_uniform", input_dim=latent_dim))
  model.add(Dense(n_outputs, activation="linear"))
  return model

# define the combined generator and discriminator model, for updating the generator
```

```
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  model.compile(loss="binary_crossentropy", optimizer ="adam")
  return model


# generate n real samples with class labels
def generate_real_samples(n):
   # generate inputs in [-0.5, 0.5]
   X1 = rand(n) - 0.5
   # generate outputs X^2
   X2 = X1 * X1
   # stack arrays
   X1 = X1.reshape(n, 1)
   X2 = X2.reshape(n, 1)
   X = hstack((X1, X2))
   # generate class labels
   y = ones((n, 1))
   return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n):
  # generate points in the latent space
  x_input = randn(latent_dim * n)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n, 1))
  return X, y

# evaluate the discriminator and plot real and fake points
def summarize_performance(epoch, generator, discriminator, latent_dim, n=100):
  # prepare real samples
  x_real, y_real = generate_real_samples(n)
  # evaluate discriminator on real examples
  _, acc_real = discriminator.evaluate(x_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(generator, latent_dim, n)
  # evaluate discriminator on fake examples
  _, acc_fake = discriminator.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print(epoch, acc_real, acc_fake)
  # scatter plot real and fake data points
  pyplot.scatter(x_real[:, 0], x_real[:, 1], color="red")
  pyplot.scatter(x_fake[:, 0], x_fake[:, 1], color="blue")
  # save plot to file
  filename = "generated_plot_e % 03d.png" % (epoch + 1)
  pyplot.savefig(filename)
  pyplot.close()

# train the generator and discriminator
def train(g_model, d_model, gan_model, latent_dim, n_epochs=10000, n_batch=128, n_eval=2000):
  # determine half the size of one batch, for updating the discriminator
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare real samples
    x_real, y_real = generate_real_samples(half_batch)
    # prepare fake examples
    x_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator
    d_model.train_on_batch(x_real, y_real)
    d_model.train_on_batch(x_fake, y_fake)
    # prepare points in latent space as input for the generator
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator"s error
    gan_model.train_on_batch(x_gan, y_gan)
    # evaluate the model every n_eval epochs
    if (i+1) % n_eval == 0:
      summarize_performance(i, g_model, d_model, latent_dim)

# size of the latent space
latent_dim = 5

# create the discriminator
discriminator = define_discriminator()

# create the generator
generator = define_generator(latent_dim)

# create the gan
gan_model = define_gan(generator, discriminator)

# train model
train(generator, discriminator, gan_model, latent_dim)
```

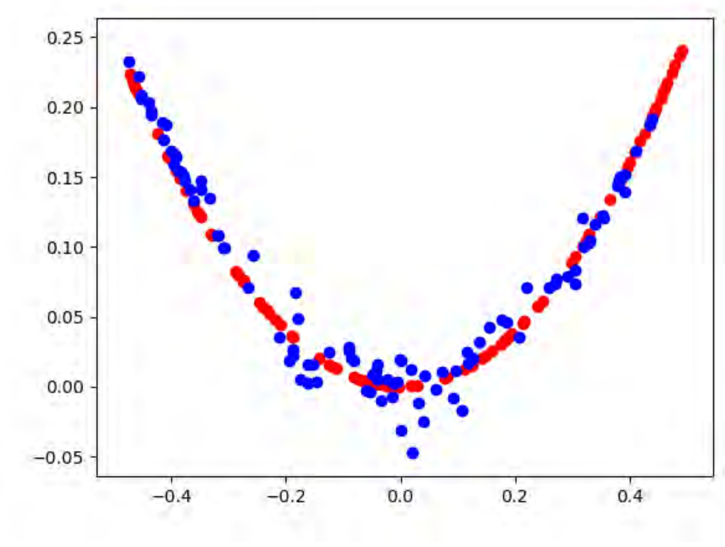
b) We can see that the training process is relatively unstable. The first column reports the iteration number, the second the classification accuracy of the discriminator for real examples, and the third column the classification accuracy of the discriminator for generated (fake) examples. In this case, we can see that the discriminator remains relatively confused about real examples, and performance on identifying fake examples varies.

```
1999 0.5 1.0
3999 0.32 0.76
5999 0.74 0.31
7999 0.54 0.43
9999 0.69 0.43
```

We will omit providing the five created plots here for brevity; instead we will look at only two. The first plot is created after 2,000 iterations and shows real (red) vs. fake (blue) samples. The model performs poorly initially with a cluster of generated points only in the positive input domain, although with the right functional relationship.



The second plot shows real (red) vs. fake (blue) after 10,000 iterations. Here we can see that the generator model does a reasonable job of generating plausible samples, with the input values in the right domain between [-0.5 and 0.5] and the output values showing the X2 relationship, or close to it.



<u>Activity wrap-up:</u>
We learn how to:
❑ Select a One Dimensional Function
❑ Define a Discriminator Model
❑ Define a Generator Model
❑ Training the Generator Model
❑ Evaluating the Performance of the GAN
❑ Complete Example of Training the GAN

Activity 5 – Develop a DCGAN for Grayscale Handwritten Digits

In this activity, we will:

- ❑ MNIST Handwritten Digit Dataset
- ❑ Define and Train the Discriminator Model
- ❑ Define and Use the Generator Model
- ❑ Training the Generator Model
- ❑ Evaluating the Performance of the GAN
- ❑ Complete Example of Training the GAN
- ❑ Using the Final Generator Model

1. MNIST Handwritten Digit Dataset:

a) The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset. It is a dataset of 70,000 small square 28 × 28 pixel grayscale images of handwritten single digits between 0 and 9.

b) Keras provides access to the MNIST dataset via the mnist.load_dataset() function. It returns two tuples, one with the input and output elements for the standard training dataset, and another with the input and output elements for the standard test dataset. The code snippet below loads the dataset and summarizes the shape of the loaded dataset.

```
# example of loading the mnist dataset
from keras.datasets.mnist import load_data
# load the images into memory
(trainX, trainy), (testX, testy) = load_data()

# summarize the shape of the dataset
print("Train", trainX.shape, trainy.shape)
print("Test", testX.shape, testy.shape)
```

Running the above loads the dataset and prints the shape of the input and output components of the train and test splits of images as shown below.

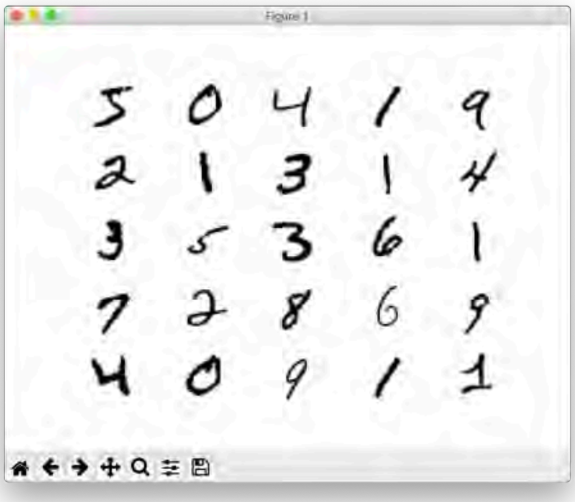
```
Train (60000, 28, 28) (60000,)
Test (10000, 28, 28) (10000,)
```

There are 60K examples in the training set and 10K in the test set and that each image is a square of 28 by 28 pixels.

c) The images are grayscale with a black background (0 pixel value) and the handwritten digits in white (pixel values near 255). The images are easier to review when we reverse the colours and plot the background as white and the handwritten digits in black. The code below plots the first 25 images from the training dataset in a 5 by 5 square.

```
# demonstrate simple x^2 function
from matplotlib import pyplot

# simple function
def calculate(x):
  return x * x

# define inputs
inputs = [-0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5]

# calculate outputs
outputs = [calculate(x) for x in inputs]

# plot the result
pyplot.plot(inputs, outputs)
pyplot.show()
```

Running the example creates a plot of 25 images from the MNIST training dataset, arranged in a 5×5 square as shown below.



d) We will use the images in the training dataset as the basis for training a Generative Adversarial Network. Specifically, the generator model will learn how to generate new plausible handwritten digits between 0 and 9, using a discriminator that will try to distinguish between real images from the MNIST training dataset and new images output by the generator model. This is a relatively simple problem that does not require a sophisticated generator or discriminator model, although it does require the generation of a grayscale output image.

2. Define and Train the Discriminator Model

a) The model must take a sample image from our dataset as input and output a classification prediction as to whether the sample is real or fake. This is a binary classification problem:.
- **Inputs**: Image with one channel and 28 × 28 pixels in size..
- **Outputs**: Binary classification, likelihood the sample is real (or fake).

b) We define the discriminator as:
   a. two convolutional layers with 64 filters each,
   b. a small kernel size of 3, and
   c. larger than normal stride of 2.
   d. the model has no pooling layers and
   e. a single node in the output layer with the sigmoid activation function to predict whether the input sample is real or fake

The model is trained to minimize the binary cross-entropy loss function, appropriate for binary classification. We will use some best practices in defining the discriminator model, such as the use of LeakyReLU instead of ReLU, using Dropout, and using the Adam version of stochastic gradient descent with a learning rate of 0.0002 and a momentum of 0.5. The function define_discriminator() below defines the discriminator model and parametrizes the size of the input image.

.

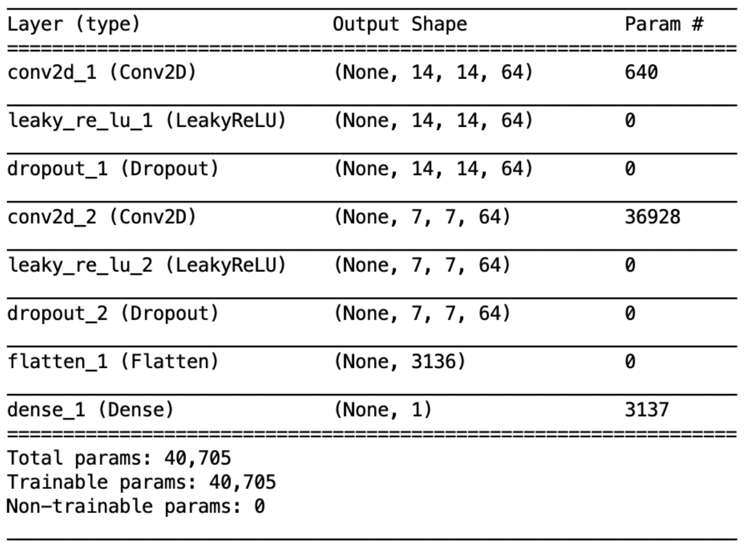
```
# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding=“same”, input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding=“same”))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation=“sigmoid”))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=“binary_crossentropy”, optimizer=opt, metrics=[“accuracy”])
  return model
```

c) The following code define the discriminator model and summarize it.

```
# example of defining the discriminator model
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Dense
from keras.layers import Conv2D
```

```python
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding="same", input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation="sigmoid"))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss="binary_crossentropy", optimizer=opt, metrics=["accuracy"])
  return model

# define model
model = define_discriminator()

# summarize the model
model.summary()

# plot the model
plot_model(model, to_file="5-3-discriminator_plot.png", show_shapes=True, show_layer_names=True)
```

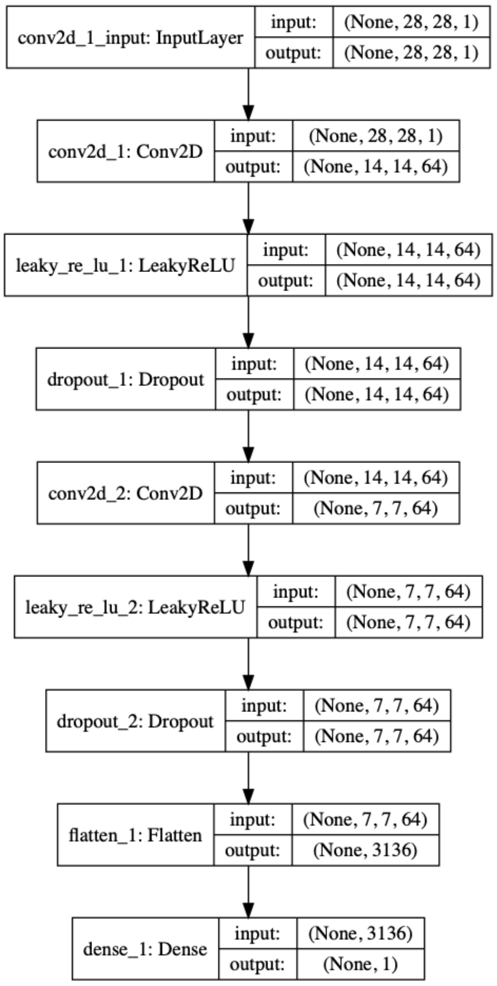
Running it summarizes the model architecture as shown below.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 14, 14, 64)        640
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 64)          36928
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 3137
=================================================================
Total params: 40,705
Trainable params: 40,705
Non-trainable params: 0
_________________________________________________________________
```

Note: We can see that the aggressive 2 × 2 stride acts to downsample the input image, first from 28 × 28 to 14 × 14, then to 7 × 7, before the model makes an output prediction. This pattern is by design as we do not use pooling layers and use the large stride as to achieve a similar downsampling effect. We will see a similar pattern, but in reverse, in the generator model in the next section.

A plot of the model is also created and we can see that the model expects two inputs and will predict a single output.



We could start training this model now with real examples with a class label of one, and randomly generated samples with a class label of zero. The development of these elements will be useful later, and it helps to see that the discriminator is just a normal neural network model for binary classification.

d) We need a function to load and prepare the dataset of real images. We will use the mnist.load data() function to load the MNIST dataset and just use the input part of the training dataset as the real images.

```
...
# load mnist dataset
(trainX, _), (_, _) = load_data()
```

e) The images are 2D arrays of pixels and convolutional neural networks expect 3D arrays of images as input, where each image has one or more channels, we must update the images to have an additional dimension for the grayscale channel. We can do this using the **expand_dims()** NumPy function and specify the final dimension for the channels-last image format.

```
...
# expand to 3d, e.g. add channels dimension
X = expand_dims(trainX, axis=-1)
```

f) Finally, we must scale the pixel values from the range of unsigned integers in [0,255] to the normalized range of [0,1]. It is best practice to use the range [-1,1], but in this case the range [0,1] works just fine.

```
# convert from unsigned ints to floats
X = X.astype('float32')
# scale from [0,255] to [0,1]
X = X / 255.0
```

g) The **load_real_samples()** function below implements this.

```
# load and prepare mnist training images
def load_real_samples():
  # load mnist dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels dimension
  X = expand_dims(trainX, axis=-1)
  # convert from unsigned ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [0,1]
   X = X / 255.0
  return X
```

h) The model will be updated in batches, specifically with a collection of real samples and a collection of generated samples. On training, an epoch is defined as one pass through the entire training dataset. We could systematically enumerate all samples in the training dataset, and that is a good approach, but good training via stochastic gradient descent requires that the training dataset be shuffled prior to each epoch. A simpler approach is to select random samples of images from the training dataset. The **generate_real_samples()** function below will take the training dataset as an argument and will select a random subsample of images; it will also return class labels for the sample, specifically a class label of 1, to indicate real images.

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate “real” class labels (1)
  y = ones((n_samples, 1))
  return X, y
```

i) Now, we need a source of fake images. We don’t have a generator model yet, so instead, we can generate images comprised of random pixel values, specifically random pixel values in the range [0,1] like our scaled real images. The **generate_fake_samples()** function below implements this behaviour and generates images of random pixel values and their associated class label of 0, for fake.

```
# generate n fake samples with class labels
def generate_fake_samples(n_samples):
  # generate uniform random numbers in [0,1]
  X = rand(28 * 28 * n_samples)
  # reshape into a batch of grayscale images
  X = X.reshape((n_samples, 28, 28, 1))
  # generate 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y
```

j) Finally, we need to train the discriminator model. This involves repeatedly retrieving samples of real images and samples of generated images and updating the model for a fixed number of iterations. We will ignore the idea of epochs for now (e.g. complete passes through the training dataset) and fit the discriminator model for a fixed number of batches. The model will learn to discriminate between real and fake (randomly generated) images rapidly, therefore, not many batches will be required before it learns to discriminate perfectly.

The **train_discriminator()** function implements this, using a batch size of 256 images where 128 are real and 128 are fake each iteration. We update the discriminator separately for real and fake examples so that we can calculate the accuracy of the model on each sample prior to the update. This gives insight into how the discriminator model is performing over time.

```
# train the discriminator model
def train_discriminator(model, dataset, n_iter=100, n_batch=256):
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_iter):
    # get randomly selected “real” samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator on real samples
    _, real_acc = model.train_on_batch(X_real, y_real)
    # generate “fake” examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update discriminator on fake samples
    _, fake_acc = model.train_on_batch(X_fake, y_fake)
    # summarize performance
    print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))
```

k) The complete code training an instance of the discriminator model on real and generated (fake) images is listed below.

```
# example of training the discriminator model on real and random mnist images
from numpy import expand_dims
from numpy import ones
from numpy import zeros
from numpy.random import rand
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import LeakyReLU

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding="same", input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation="sigmoid"))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss="binary_crossentropy", optimizer=opt, metrics=["accuracy"])
  return model

# load and prepare mnist training images
def load_real_samples():
  # load mnist dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels dimension
  X = expand_dims(trainX, axis=-1)
  # convert from unsigned ints to floats
  X = X.astype("float32")
  # scale from [0,255] to [0,1]
  X = X / 255.0
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate "real" class labels (1)
  y = ones((n_samples, 1))
  return X, y

# generate n fake samples with class labels
def generate_fake_samples(n_samples):
  # generate uniform random numbers in [0,1]
  X = rand(28 * 28 * n_samples)
  # reshape into a batch of grayscale images
  X = X.reshape((n_samples, 28, 28, 1))
  # generate "fake" class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# train the discriminator model
def train_discriminator(model, dataset, n_iter=100, n_batch=256):
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_iter):
    # get randomly selected "real" samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator on real samples
    _, real_acc = model.train_on_batch(X_real, y_real)
    # generate "fake" examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update discriminator on fake samples
    _, fake_acc = model.train_on_batch(X_fake, y_fake)
    # summarize performance
    print(" > %d real=%.0f%% fake=%.0f%%" % (i + 1, real_acc * 100, fake_acc * 100))

# define the discriminator model
model = define_discriminator()

# load image data
dataset = load_real_samples()

# fit the model
train_discriminator(model, dataset)
```

The discriminator model learns to tell the difference between real and generated MNIST images very quickly, in about 30 interations.

```
> 92 real=100% fake=100%
> 93 real=100% fake=100%
> 94 real=100% fake=100%
> 95 real=100% fake=100%
> 96 real=100% fake=100%
> 97 real=100% fake=100%
> 98 real=100% fake=100%
> 99 real=100% fake=100%
> 100 real=100% fake=100%
```

3. Define and Use the Generator Model

a) The generator model is responsible for creating new, fake but plausible images of handwritten digits. It does this by taking a point from the latent space as input and outputting a square grayscale image. The latent space is an arbitrarily defined vector space of Gaussian-distributed values, e.g. 100 dimensions. It has no meaning, but by drawing points from this space randomly and providing them to the generator model during training, the generator model will assign meaning to the latent points. At the end of training, the latent vector space represents a compressed representation of the output space, MNIST images, that only the generator knows how to turn into plausible MNIST images.

- **Inputs**: Point in latent space, e.g. a 100 element vector of Gaussian random numbers.
- **Outputs**: Two-dimensional square grayscale image of 28 × 28 pixels with pixel values in [0,1].

Note:

1) We don't have to use a 100 element vector as input; it is a round number and widely used, but expect that 10, 50, or 500 would work just as well.
2) Developing a generator model requires that we transform a vector from the latent space with, 100 dimensions to a 2D array with 28 × 28 or 784 values. There are a number of ways to achieve this but there is one approach that has proven effective at deep convolutional generative adversarial networks. It involves two main elements. The first is a Dense layer as the first hidden layer that has enough nodes to represent a low-resolution version of the output image. Specifically, an image half the size (one quarter the area) of the output image would be 14 × 14 or 196 nodes, and an image one quarter the size (one eighth the area) would be 7 × 7 or 49 nodes.

b) We don't just want one low-resolution version of the image; we want many parallel versions or interpretations of the input. This is a pattern in convolutional neural networks where we have many parallel filters resulting in multiple parallel activation maps, called feature maps, with different interpretations of the input. We want the same thing in reverse: many parallel versions of our output with different learned features that can be collapsed in the output layer into a final image. The model needs space to invent, create, or generate. Therefore, **the first hidden layer, the Dense layer needs enough nodes for multiple low-resolution versions of our output image, such as 128**.

```
...
# foundation for 7x7 image
model.add(Dense(128 * 7 * 7, input_dim=100))
```

c) The activations from these nodes can then be reshaped into something image-like to pass into a convolutional layer, such as 128 different 7 × 7 feature maps.

```
...
model.add(Reshape((7, 7, 128)))
```

d) The next major architectural innovation involves upsampling the low-resolution image to a higher resolution version of the image. There are two common ways to do this upsampling process, sometimes called deconvolution. One way is to use an UpSampling2D layer (like a reverse pooling layer) followed by a normal Conv2D layer. The other and perhaps more modern way is to combine these two operations into a single layer, called a Conv2DTranspose. We will use this latter approach for our generator.
The **Conv2DTranspose** layer can be configured with a stride of (2 × 2) that will quadruple the area of the input feature maps (double their width and height dimensions). It is also good practice to use a kernel size that is a factor of the stride (e.g. double) to avoid a checkerboard pattern that can be observed when upsampling.

```
...
# upsample to 14x14
model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding=✬same✬))
```

e) This can be repeated to arrive at our 28×28 output image. we will use the LeakyReLU activation with a default slope of 0.2, reported as a best practice when training GAN models. The output layer of the model is a Conv2D with one filter and a kernel size of 7 × 7 and 'same' padding, designed to create a single feature map and preserve its dimensions at 28 × 28 pixels. A sigmoid activation is used to ensure output values are in the desired range of [0,1]. The define generator() function below implements this and defines the generator model. The generator model is not compiled and does not specify a loss function or optimization algorithm. This is because the generator is not trained directly

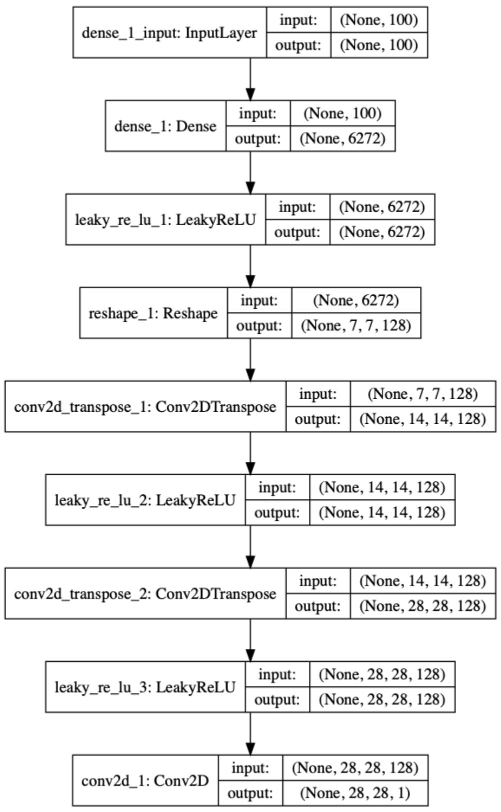
f) The follow code define the generator model and summarize it.

```
# example of defining the generator model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4, 4), strides=(2, 2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4, 4), strides=(2, 2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7, 7), activation="sigmoid", padding ="same"))
  return model

# define the size of the latent space
latent_dim = 100

# define the generator model
model = define_generator(latent_dim)

# summarize the model
model.summary()

# plot the model
plot_model(model, to_file="generator_plot.png", show_shapes = True, show_layer_names = True)
```

Running the example summarizes the layers of the model and their output shape. We can see that, as designed, the first hidden layer has 6,272 parameters or 128 × 7 × 7, the activations of which are reshaped into 128 7 × 7 feature maps. The feature maps are then upscaled via the two Conv2DTranspose layers to the desired output shape of 28 × 28, until the output layer, where a single activation map is output.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 6272)              633472
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 6272)              0
_________________________________________________________________
reshape_1 (Reshape)          (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 14, 14, 128)       262272
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 28, 28, 128)       262272
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 28, 28, 128)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 1)         6273
=================================================================
Total params: 1,164,289
Trainable params: 1,164,289
Non-trainable params: 0
_________________________________________________________________
```

A plot of the model is also created and we can see that the model expects two inputs and will predict a single output.


dense_1_input: InputLayer
input: (None, 100)
output: (None, 100)
dense_1: Dense
input: (None, 100)
output: (None, 6272)
leaky_re_lu_1: LeakyReLU
input: (None, 6272)
output: (None, 6272)
reshape_1: Reshape
input: (None, 6272)
output: (None, 7, 7, 128)
conv2d_transpose_1: Conv2DTranspose
input: (None, 7, 7, 128)
output: (None, 14, 14, 128)
leaky_re_lu_2: LeakyReLU
input: (None, 14, 14, 128)
output: (None, 14, 14, 128)
conv2d_transpose_2: Conv2DTranspose
input: (None, 14, 14, 128)
output: (None, 28, 28, 128)
leaky_re_lu_3: LeakyReLU
input: (None, 28, 28, 128)
output: (None, 28, 28, 128)
conv2d_1: Conv2D
input: (None, 28, 28, 128)
output: (None, 28, 28, 1)

This model cannot do much at the moment. Nevertheless, we can demonstrate how to use it to generate samples. This is a helpful demonstration to understand the generator as just another model, and some of these elements will be useful later.

g) The first step is to draw new points from the latent space. We can achieve this by calling the **randn()** NumPy function for generating arrays of random numbers drawn from a standard Gaussian. The array of random numbers can then be reshaped into samples, that is n rows with 100 elements per row. The **generate_latent_points()** function below implements this and generates the desired number of points in the latent space that can be used as input to the generator model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

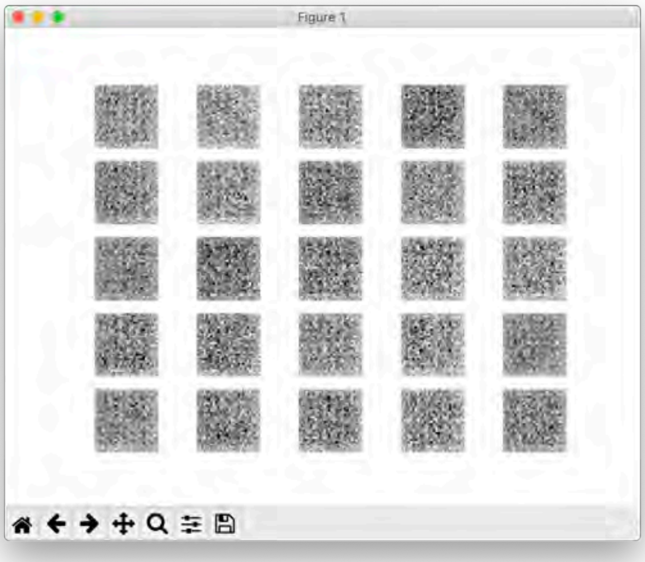
h) Next, we can use the generated points as input to the generator model to generate new samples, then plot the samples. We can update the **generate_fake_samples()** function from the previous section to take the generator model as an argument and use it to generate the desired number of samples by first calling the **generate_latent_points()** function to generate the required number of points in latent space as input to the model. The updated **generate_fake_samples()** function is listed below and returns both the generated samples and the associated class labels.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create ❀fake❀ class labels (0)
  y = zeros((n_samples, 1))
  return X, y
```

i) The complete code of generating new MNIST images with the untrained generator model is listed below.

```
# example of defining and using the generator model
from numpy import zeros
from numpy.random import randn
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from matplotlib import pyplot

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4, 4), strides=(2, 2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4, 4), strides=(2, 2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7, 7), activation="sigmoid", padding ="same"))
  return model

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create "fake" class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# size of the latent space
latent_dim = 100

# define the discriminator model
model = define_generator(latent_dim)

# generate samples
n_samples = 25
X, _ = generate_fake_samples(model, latent_dim, n_samples)

# plot the generated samples
for i in range(n_samples):
  # define subplot
  pyplot.subplot(5, 5, 1 + i)
  # turn off axis labels
  pyplot.axis("off")
  # plot single image
  pyplot.imshow(X[i, :, :, 0], cmap="gray_r")

# show the figure
pyplot.show()
```

Running the example generates 25 examples of fake MNIST images and visualizes them on a single plot of 5 by 5 images. As the model is not trained, the generated images are completely random pixel values in [0, 1].



4. Training the Generator Model

a) The weights in the generator model are updated based on the performance of the discriminator model. When the discriminator is good at detecting fake samples, the generator is updated more, and when the discriminator model is relatively poor or confused when detecting fake samples, the generator model is updated less. This defines the zero-sum or adversarial relationship between these two models. There may be many ways to implement this using the Keras API, but perhaps the simplest approach is to create a new model that combines the generator and discriminator models.

A new GAN model can be defined that stacks the generator and discriminator such that the generator receives as input random points in the latent space and generates samples that are fed into the discriminator model directly, classified, and the output of this larger model can be used to update the model weights of the generator.

To be clear, we are not talking about a new third model, just a new logical model that uses the already-defined layers and weights from the standalone generator and discriminator models. Only the discriminator is concerned with distinguishing between real and fake examples, therefore the discriminator model can be trained in a standalone manner.

The generator model is only concerned with the discriminator's performance on fake examples. Therefore, we will mark all of the layers in the discriminator as not trainable when it is part of the GAN model so that they cannot be updated and overtrained on fake examples. When training the generator via this logical GAN model, there is one more important change. We want the discriminator to think that the samples output by the generator are real, not fake. Therefore, when the generator is trained as part of the GAN model, we will mark the generated samples as real (class = 1).

- **Inputs**: Point in latent space, e.g. a 100 element vector of Gaussian random numbers.
- **Outputs**: Binary classification, likelihood the sample is real (or fake)..

b) The **define_gan()** function below takes as arguments the already-defined generator and discriminator models and creates the new logical third model subsuming these two models. The weights in the discriminator are marked as not trainable, which only affects the weights as seen by the GAN model and not the standalone discriminator model. The GAN model then uses the same binary cross-entropy loss function as the discriminator and the efficient Adam version of stochastic gradient with the learning rate of 0.0002 and momentum 0.5, recommended when training deep convolutional GANs. .

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5) model.compile(loss="binary_crossentropy", optimizer=opt)
  return model
```

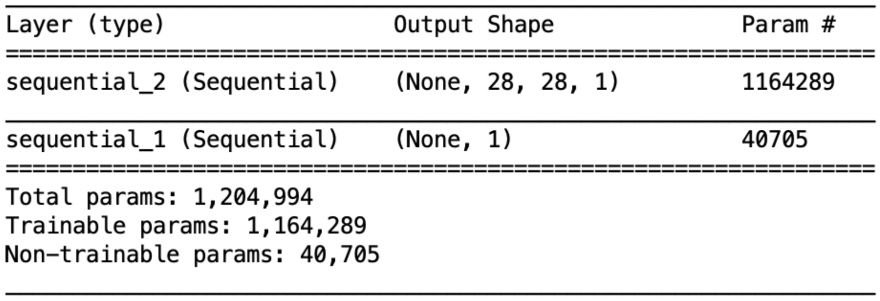
Note: Making the discriminator not trainable is a clever trick in the Keras API. The trainable property impacts the model when it is compiled. The discriminator model was compiled with trainable layers, therefore the model weights in those layers will be updated when the standalone model is updated via calls to **train_on_batch()**. The discriminator model was marked as not trainable, added to the GAN model, and compiled. In this model, the model weights of the discriminator model are not trainable and cannot be changed when the GAN model is updated via calls to **train_on_batch()**.

c) The complete example of creating the discriminator, generator, and composite model is listed below.

```
# demonstrate creating the three models in the gan
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding="same", input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation="sigmoid"))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss="binary_crossentropy", optimizer=opt, metrics=["accuracy"])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation="sigmoid", padding="same"))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss="binary_crossentropy", optimizer=opt)
  return model

# size of the latent space
latent_dim = 100

# create the discriminator
d_model = define_discriminator()

```

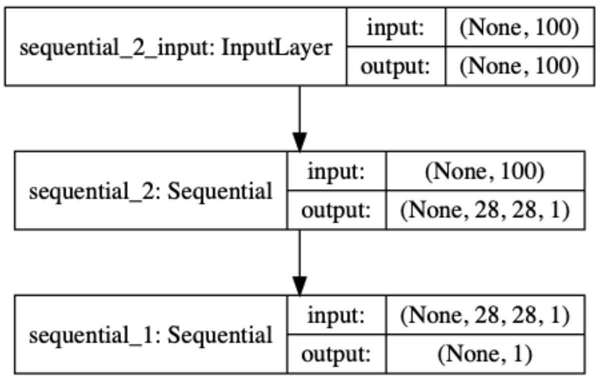
```python
# create the generator
g_model = define_generator(latent_dim)

# create the gan
gan_model = define_gan(g_model, d_model)

# summarize gan model
gan_model.summary()

# plot gan model
plot_model(gan_model, to_file="5-7-gan_plot.png", show_shapes=True, show_layer_names=True)
```

Running the example first creates a summary of the composite model. We can see that the model expects MNIST images as input and predicts a single value as output.



```
Layer (type)                 Output Shape              Param #
=================================================================
sequential_2 (Sequential)    (None, 28, 28, 1)         1164289
_________________________________________________________________
sequential_1 (Sequential)    (None, 1)                 40705
=================================================================
Total params: 1,204,994
Trainable params: 1,164,289
Non-trainable params: 40,705
_________________________________________________________________
```

A plot of the model is also created and we can see that the model expects a 100-element point in latent space as input and will predict a single output classification label.



d) Training the composite model involves generating a batch-worth of points in the latent space via the **generate_latent_points()** function in the previous section, and class = 1 labels and calling the **train_on_batch()** function. The **train_gan()** function below demonstrates this, although it is pretty simple as only the generator will be updated each epoch, leaving the discriminator with default model weights.

```python
# train the composite model
def train_gan(gan_model, latent_dim, n_epochs=10000, n_batch=128):
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare points in latent space as input for the generator
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator’s error
    gan_model.train_on_batch(x_gan, y_gan)
```

e) Instead, what is required is that we first update the discriminator model with real and fake samples, then update the generator via the composite model. This requires combining elements from the **train_discriminator()** function defined in the discriminator section and the **train_gan()** function defined above. It also requires that we enumerate over both epochs and batches within in an epoch. The complete train function for updating the discriminator model and the generator (via the composite model) is listed below. Note:

1) the number of batches within an epoch is defined by how many times the batch size divides into the training dataset. We have a dataset size of 60K samples and a batch size of 256, so with rounding down, there are 60000/256 or 234 batches per epoch.
2) The discriminator model is updated once per batch by combining one half a batch (128) of fake and real (128) examples into a single batch via the vstack() NumPy function. You could update the discriminator with each half batch separately (recommended for more complex datasets) but combining the samples into a single batch will be faster over a long run, especially when training on GPU hardware.

3) Finally, we report the loss for each batch. It is critical to keep an eye on the loss over batches. The reason for this is that a crash in the discriminator loss indicates that the generator model has started generating rubbish examples that the discriminator can easily discriminate. Monitor the discriminator loss and expect it to hover around 0.5 to 0.8 per batch on this dataset. The generator loss is less critical and may hover between 0.5 and 2 or higher on this dataset. A clever programmer might even attempt to detect the crashing loss of the discriminator, halt, and then restart the training process.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected ✯real✯ samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # generate ✯fake✯ examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # create training set for the discriminator
      X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))
      # update discriminator model weights
      d_loss, _ = d_model.train_on_batch(X, y)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator✯s error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print(✯>%d, %d/%d, d=%.3f, g=%.3f✯ % (i+1, j+1, bat_per_epo, d_loss, g_loss))
```

5. Evaluating the Performance of the GAN

a) Generally**, there are no objective ways to evaluate the performance of a GAN model**. We cannot calculate this objective error score for generated images.

b) It might be possible in the case of MNIST images because the images are so well constrained, but in general, it is not possible (yet). Instead, images must be subjectively evaluated for quality by a human operator. This means that we cannot know when to stop training without looking at examples of generated images. In turn, the adversarial nature of the training process means that the generator is changing after every batch, meaning that once good enough images can be generated, the subjective quality of the images may then begin to vary, improve, or even degrade with subsequent updates. There are three ways to handle this complex training situation.

- Periodically evaluate the classification accuracy of the discriminator on real and fake images.
- Periodically generate many images and save them to file for subjective review.
- Periodically save the generator model.

All three of these actions can be performed at the same time for a given training epoch, such as every five or 10 training epochs. The result will be a saved generator model for which we have a way of subjectively assessing the quality of its output and objectively knowing how well the discriminator was fooled at the time the model was saved. Training the GAN over many epochs, such as hundreds or thousands of epochs, will result in many snapshots of the model that can be inspected and from which specific outputs and models can be cherry-picked for later use.

c) First, we can define a function called **summarize_performance()** that will summarize the performance of the discriminator model. It does this by retrieving a sample of real MNIST images, as well as generating the same number of fake MNIST images with the generator model, then evaluating the classification accuracy of the discriminator model on each sample and reporting these scores.

```
# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print(“>Accuracy real: %.0f%%, fake: %.0f%%” % (acc_real*100, acc_fake*100))
```

d) This function can be called from the **train()** function based on the current epoch number, such as every 10 epochs.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
  ...
  # evaluate the model performance, sometimes
  if (i+1) % 10 == 0:
    summarize_performance(i, g_model, d_model, dataset, latent_dim)
```

e) Next, we can update the **summarize_performance()** function to both save the model and to create and save a plot generated examples. The generator model can be saved by calling the save() function on the generator model and providing a unique filename based on the training epoch number.

```
...
# save the generator model tile file
filename = “generator_model_%03d.h5” % (epoch + 1)
g_model.save(filename)
```

f) We can develop a function to create a plot of the generated samples. As we are evaluating the discriminator on 100 generated MNIST images, we can plot all 100 images as a 10 by 10 grid. The save_plot() function below implements this, again saving the resulting plot with a unique filename based on the epoch number.

```
# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, epoch, n=10):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis(“off”)
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap=”gray_r”)

# save plot to file
filename = “generated_plot_e%03d.png” % (epoch+1)
pyplot.savefig(filename)
pyplot.close()
```

g) The updated summarize performance() function with these additions is listed below.

```
# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print(“>Accuracy real: %.0f%%, fake: %.0f%%” % (acc_real*100, acc_fake*100))

  # save plot
  save_plot(x_fake, epoch)
  # save the generator model tile file
  filename = ✯generator_model_%03d.h5✯% (epoch + 1)
  g_model.save(filename)
```

6. Complete Example of Training the GAN

a) We now have everything we need to train and evaluate a GAN on our chosen one-dimensional function. The complete example is listed below.

```
# example of training a gan on mnist
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy import vstack
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding="same", input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation="sigmoid"))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss="binary_crossentropy", optimizer=opt, metrics=["accuracy"])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding="same"))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation="sigmoid", padding="same"))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss="binary_crossentropy", optimizer=opt)
  return model

# load and prepare mnist training images
def load_real_samples():
  # load mnist dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels dimension
  X = expand_dims(trainX, axis=-1)
  # convert from unsigned ints to floats
  X = X.astype("float32")
  # scale from [0,255] to [0,1]
  X = X / 255.0
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate "real" class labels (1)
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create "fake" class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, epoch, n=10):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis("off")
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap="gray_r")

  # save plot to file
  filename = "generated_plot_e%03d.png" % (epoch + 1)
  pyplot.savefig(filename)
  pyplot.close()

# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print(" > Accuracy real: %.0f%%, fake:%.0f%%" % (acc_real * 100, acc_fake * 100))
  # save plot
  save_plot(x_fake, epoch)
  # save the generator model tile file
  filename = "generator_model_%03d.h5" % (epoch + 1)
  g_model.save(filename)

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected "real" samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # generate "fake" examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # create training set for the discriminator
      X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))
      # update discriminator model weights
      d_loss, _ = d_model.train_on_batch(X, y)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator"s error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print(">%d, %d/%d, d=%.3f, g=%.3f" % (i + 1, j + 1, bat_per_epo, d_loss, g_loss))
    # evaluate the model performance, sometimes
    if (i + 1) % 10 == 0:
      summarize_performance(i, g_model, d_model, dataset, latent_dim)

# size of the latent space
latent_dim = 100

# create the discriminator
d_model = define_discriminator()

```

```
# create the generator
g_model = define_generator(latent_dim)

# create the gan
gan_model = define_gan(g_model, d_model)

# load image data
dataset = load_real_samples()

# train model
train(g_model, d_model, gan_model, dataset, latent_dim)
```

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible.

The chosen configuration results in the stable training of both the generative and discriminative model. The model performance is reported every batch, including the loss of both the discriminative (d) and generative (g) models.

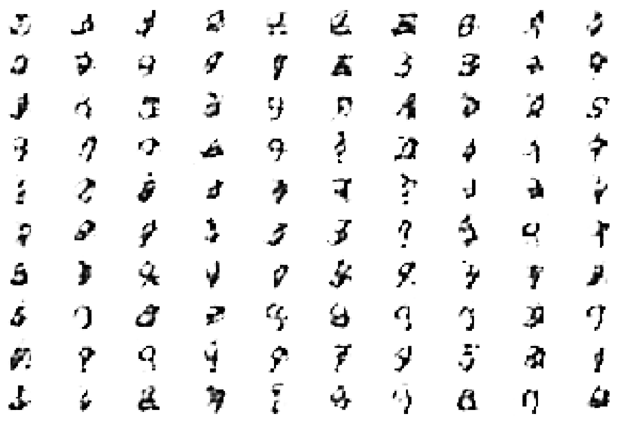
Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the loss remains stable over the course of training.

```
>1, 1/234, d=0.711, g=0.678
>1, 2/234, d=0.703, g=0.698
>1, 3/234, d=0.694, g=0.717
>1, 4/234, d=0.684, g=0.740
>1, 5/234, d=0.679, g=0.757
>1, 6/234, d=0.668, g=0.777
...
>100, 230/234, d=0.690, g=0.710
>100, 231/234, d=0.692, g=0.705
>100, 232/234, d=0.698, g=0.701
>100, 233/234, d=0.697, g=0.688
>100, 234/234, d=0.693, g=0.698
```

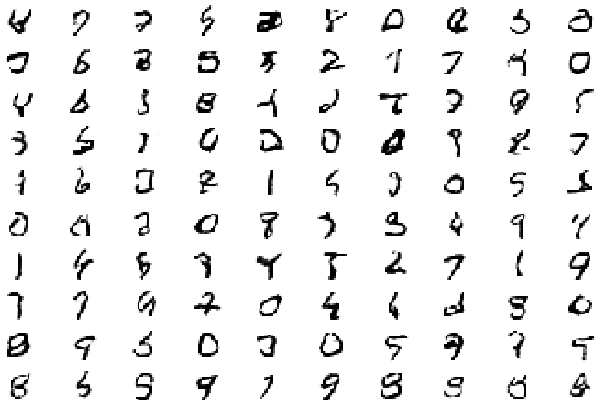
The generator is evaluated every 10 epochs, resulting in 10 evaluations, 10 plots of generated images, and 10 saved models. In this case, we can see that the accuracy fluctuates over training. When viewing the discriminator model's accuracy score in concert with generated images, we can see that the accuracy on fake examples does not correlate well with the subjective quality of images, but the accuracy for real examples may. It is crude and possibly unreliable metric of GAN performance, along with loss.

More training, beyond some point, does not mean better quality generated images. In this case, the results after 10 epochs are low quality, although we can see that the generator has learned to generate centred figures in white on a black background



After 20 or 30 more epochs, the model begins to generate very plausible MNIST figures, suggesting that 100 epochs are probably not required for the chosen model configurations.

The generated images after 100 epochs are not greatly different, but I believe we can detect less blocky-ness in the curves.



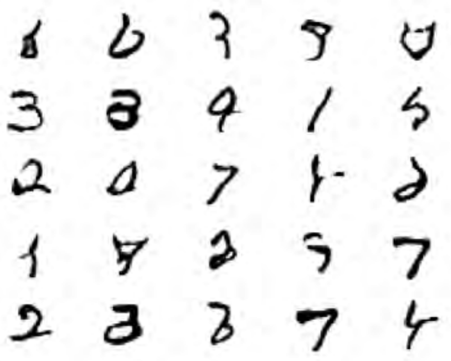
7. Using the Final Generator Model

a) Once a final generator model is selected, it can be used in a standalone manner for your application. This involves first loading the model from file, then using it to generate images. The generation of each image requires a point in the latent space as input. The complete example of loading the saved model and generating images is listed below. In this case, we will use the model saved after 100 training epochs, but the model saved after 40 or 50 epochs would work just as well.

```
# example of loading the generator model and generating images
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis("off")
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap="gray_r")
  pyplot.show()

# load model
```

```
model = load_model("generator_model_100.h5")

# generate images
latent_points = generate_latent_points(100, 25) # generate images
X = model.predict(latent_points)

# plot the result
save_plot(X, 5)
```

Running the example first loads the model, samples 25 random points in the latent space, generates 25 images, then plots the results as a single image.

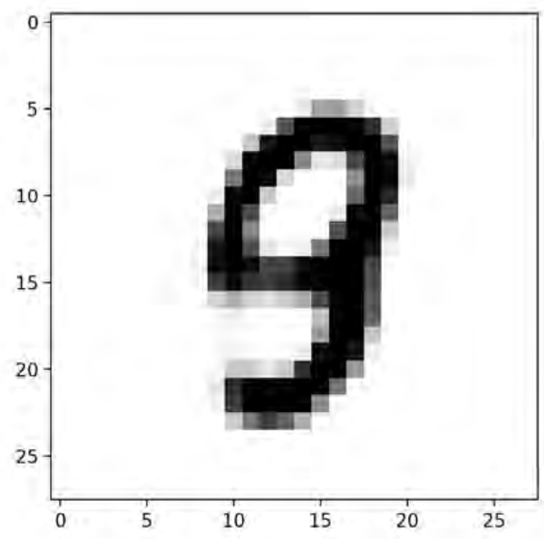


In this case, we can see that most of the images are plausible, or plausible pieces of handwritten digits.

b) The latent space now defines a compressed representation of MNIST handwritten digits. You can experiment with generating different points in this space and see what types of numbers they generate. The example below generates a single handwritten digit using a vector of all 0.0 values.

```
# example of generating an image for a specific point in the latent space
from keras.models import load_model
from numpy import asarray
from matplotlib import pyplot

# load model
model = load_model("generator_model_100.h5")
# all 0s
vector = asarray([[0.0 for _ in range(100)]])
# generate image
X = model.predict(vector)
# plot the result
pyplot.imshow(X[0, :, :, 0], cmap="gray_r")
pyplot.show()
```

In this case, a vector of all zeros results in a handwritten 9. You can then try navigating the space and see if you can generate a range of similar, but different handwritten digits.



<u>Activity wrap-up:</u>
We learn how to:
- ❑ MNIST Handwritten Digit Dataset
- ❑ Define and Train the Discriminator Model
- ❑ Define and Use the Generator Model
- ❑ Training the Generator Model
- ❑ Evaluating the Performance of the GAN
- ❑ Complete Example of Training the GAN
- ❑ Using the Final Generator Model