Exploratory Configuration of a Multilayer Perceptron Network for Time Series Forecasting

by **Jason Brownlee** on April 26, 2017 in **Deep Learning**

It can be difficult when starting out on a new predictive modeling project with neural networks.

There is so much to configure, and no clear idea where to start.

It is important to be systematic. You can break bad assumptions and quickly hone in on configurations that work and areas for further investigation likely to payoff.

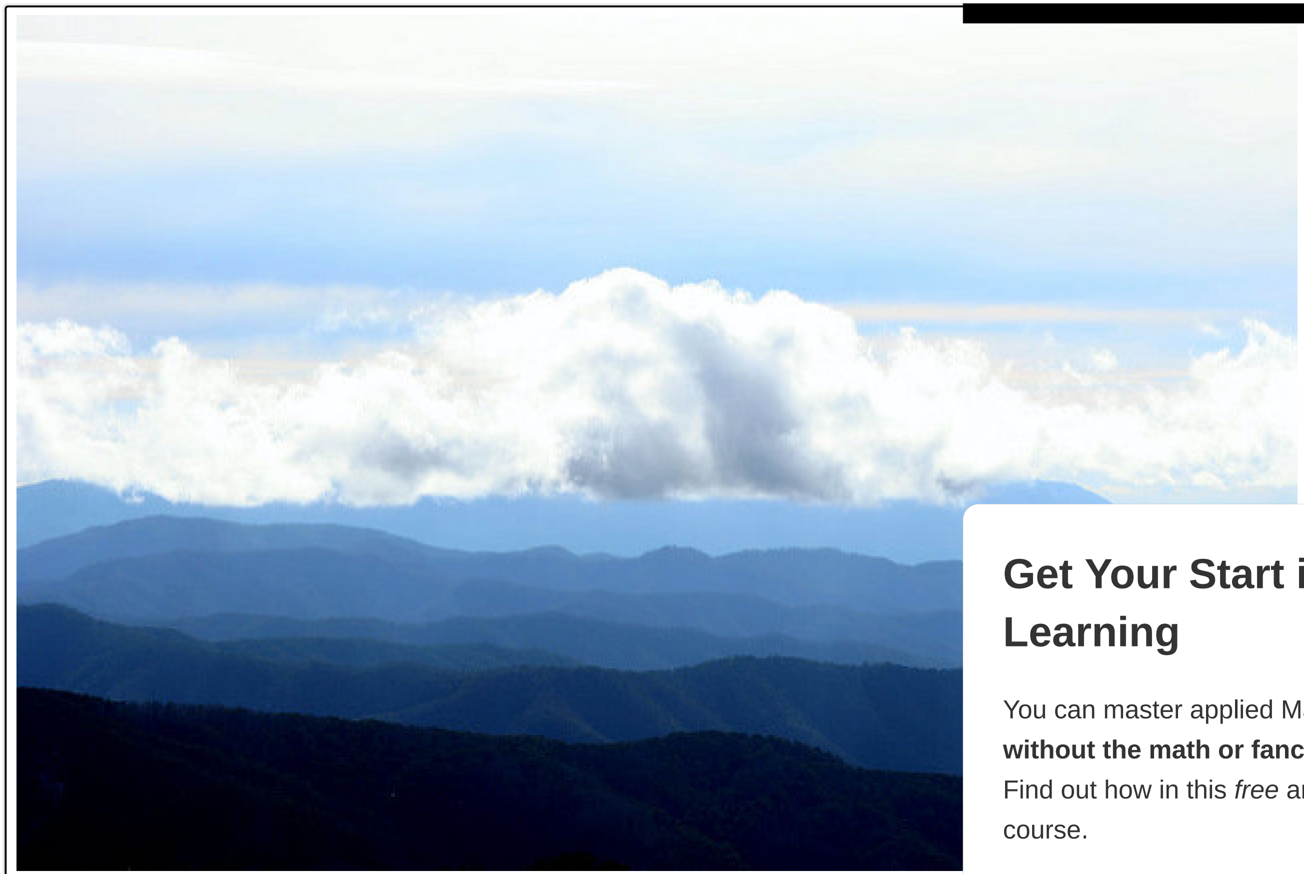
In this tutorial, you will discover how to use exploratory configuration of multilayer perceptron (MLP) [redacted]me series forecasting.

After completing this tutorial, you will know:

- How to design a robust experimental test harness to evaluate MLP models for time series forecasting.
- Systematic experimental designs for varying epochs, neurons, and lag configurations.
- How to interpret results and use diagnostics to learn more about well-performing models.

Let's get started.

- **Update July/2017**: Changed function for creating models to be more descriptive.



Exploratory Configuration of a Multilayer Perceptron Network for Time Ser
Photo by Lachlan Donald, some rights reserved.

Tutorial Overview

This tutorial is broken down into 6 parts. They are:

1. Shampoo Sales Dataset
2. Experimental Test Harness
3. Vary Training Epochs
4. Vary Hidden Layer Neurons
5. Vary Hidden Layer Neurons with Lag
6. Review of Results

Environment

This tutorial assumes you have a Python SciPy environment installed. You can use either Python 2 or 3 with this example.

This tutorial assumes you have Keras v2.0 or higher installed with either the TensorFlow or Theano backend.

This tutorial also assumes you have scikit-learn, Pandas, NumPy, and Matplotlib installed.

Next, let's take a look at a standard time series forecasting problem that we can use as context for this experiment.

If you need help setting up your Python environment, see this post:

- How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda

Shampoo Sales Dataset

This dataset describes the monthly number of sales of shampoo over a 3-year period.

The units are a sales count and there are 36 observations. The original dataset is credited to Makrid

You can download and learn more about the dataset here.

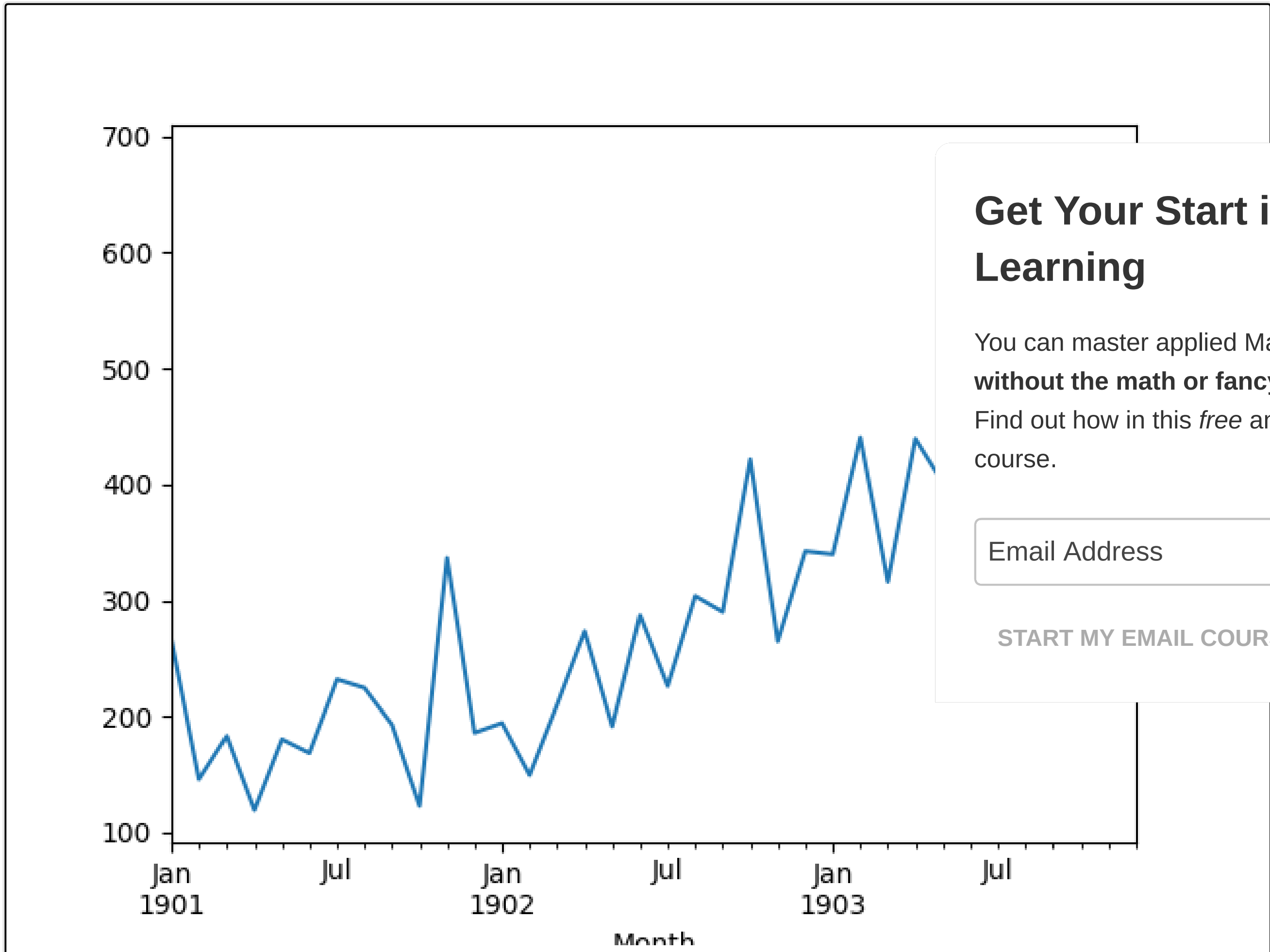
The example below loads and creates a plot of the loaded dataset.

```
# load and plot dataset
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
# load dataset
def parser(x):
    return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=True, date_parser=parser)
# summarize first few rows
print(series.head())
# line plot
series.plot()
pyplot.show()
```

Running the example loads the dataset as a Pandas Series and prints the first 5 rows.

```
Month
1901-01-01 266.0
1901-02-01 145.9
1901-03-01 183.1
1901-04-01 119.3
1901-05-01 180.3
Name: Sales, dtype: float64
```

A line plot of the series is then created showing a clear increasing trend.



Line Plot of Shampoo Sales Dataset

Next, we will take a look at the model configuration and test harness used in the experiment.

Experimental Test Harness

This section describes the test harness used in this tutorial.

Data Split

We will split the Shampoo Sales dataset into two parts: a training and a test set.

The first two years of data will be taken for the training dataset and the remaining one year of data will be used for the test set.

Models will be developed using the training dataset and will make predictions on the test dataset.

The persistence forecast (naive forecast) on the test dataset achieves an error of 136.761 monthly s
bound of performance on the test set.

Model Evaluation

A rolling-forecast scenario will be used, also called walk-forward model validation.

Each time step of the test dataset will be walked one at a time. A model will be used to make a forec ue
from the test set will be taken and made available to the model for the forecast on the next time step

This mimics a real-world scenario where new Shampoo Sales observations would be available each g
month.

This will be simulated by the structure of the train and test datasets.

All forecasts on the test dataset will be collected and an error score calculated to summarize the skill of the model. The root mean squared error (RMSE) will be used as it punishes large errors and results in a score that is in the same units as the forecast data, namely monthly shampoo sales.

Data Preparation

Before we can fit an MLP model to the dataset, we must transform the data.

The following three data transforms are performed on the dataset prior to fitting a model and making a forecast.

1. **Transform the time series data so that it is stationary**. Specifically, a lag=1 differencing to remove the increasing trend in the data.
2. **Transform the time series into a supervised learning problem**. Specifically, the organization of data into input and output patterns where the observation at the previous time step is used as an input to forecast the observation at the current timestep
3. **Transform the observations to have a specific scale**. Specifically, to rescale the data to values between -1 and 1.

These transforms are inverted on forecasts to return them into their original scale before calculating and error score.

MLP Model

We will use a base MLP model with 1 neuron hidden layer, a rectified linear activation function on hid tput neurons.

A batch size of 4 is used where possible, with the training data truncated to ensure the number of pa e of 2 is used.

Normally, the training dataset is shuffled after each batch or each epoch, which can aid in fitting the t problems. Shuffling was turned off for all experiments as it seemed to result in better performance. M me series forecasting.

The model will be fit using the efficient ADAM optimization algorithm and the mean squared error los

Experimental Runs

Each experimental scenario will be run 30 times and the RMSE score on the test set will be recorded from the end each run.

Let's dive into the experiments.

Vary Training Epochs

In this first experiment, we will investigate varying the number of training epochs for a simple MLP w[redacted] layer.

We will use a batch size of 4 and evaluate training epochs 50, 100, 500, 1000, and 2000.

The complete code listing is provided below.

This code listing will be used as the basis for all following experiments, with only the changes to this code provided in subsequent sections.

```
from pandas import DataFrame
from pandas import Series
from pandas import concat
from pandas import read_csv
from pandas import datetime
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense
from math import sqrt
import matplotlib
# be able to save images on server
matplotlib.use('Agg')
from matplotlib import pyplot
import numpy

# date-time parsing function for loading the dataset
def parser(x):
    return datetime.strptime('190'+x, '%Y-%m')

# frame a sequence as a supervised learning problem
def timeseries_to_supervised(data, lag=1):
    df = DataFrame(data)
    columns = [df.shift(i) for i in range(1, lag+1)]
    columns.append(df)
    df = concat(columns, axis=1)
    return df

# create a differenced series
def difference(dataset, interval=1):
    diff = list()
    for i in range(interval, len(dataset)):
        value = dataset[i] - dataset[i - interval]
        diff.append(value)
    return Series(diff)
```

```

# invert differenced value
def inverse_difference(history, yhat, interval=1):
    return yhat + history[-interval]

# scale train and test data to [-1, 1]
def scale(train, test):
    # fit scaler
    scaler = MinMaxScaler(feature_range=(-1, 1))
    scaler = scaler.fit(train)
    # transform train
    train = train.reshape(train.shape[0], train.shape[1])
    train_scaled = scaler.transform(train)
    # transform test
    test = test.reshape(test.shape[0], test.shape[1])
    test_scaled = scaler.transform(test)
    return scaler, train_scaled, test_scaled

# inverse scaling for a forecasted value
def invert_scale(scaler, X, yhat):
    new_row = [x for x in X] + [yhat]
    array = numpy.array(new_row)
    array = array.reshape(1, len(array))
    inverted = scaler.inverse_transform(array)
    return inverted[0, -1]

# fit an MLP network to training data
def fit_model(train, batch_size, nb_epoch, neurons):
    X, y = train[:, 0:-1], train[:, -1]
    model = Sequential()
    model.add(Dense(neurons, activation='relu', input_dim=X.shape[1]))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(X, y, epochs=nb_epoch, batch_size=batch_size, verbose=0, shuffle=False)
    return model

# run a repeated experiment
def experiment(repeats, series, epochs, lag, neurons):
    # transform data to be stationary
    raw_values = series.values
    diff_values = difference(raw_values, 1)
    # transform data to be supervised learning
    supervised = timeseries_to_supervised(diff_values, lag)
    supervised_values = supervised.values[lag:,:]
    # split data into train and test-sets
    train, test = supervised_values[0:-12], supervised_values[-12:]
    # transform the scale of the data
```

```
    scaler, train_scaled, test_scaled = scale(train, test)
    # run experiment
    error_scores = list()
    for r in range(repeats):
        # fit the model
        batch_size = 4
        train_trimmed = train_scaled[2:, :]
        model = fit_model(train_trimmed, batch_size, epochs, neurons)
        # forecast test dataset
        test_reshaped = test_scaled[:,0:-1]
        output = model.predict(test_reshaped, batch_size=batch_size)
        predictions = list()
        for i in range(len(output)):
            yhat = output[i,0]
            X = test_scaled[i, 0:-1]
            # invert scaling
            yhat = invert_scale(scaler, X, yhat)
            # invert differencing
            yhat = inverse_difference(raw_values, yhat, len(test_scaled)+1-i)
            # store forecast
            predictions.append(yhat)
        # report performance
        rmse = sqrt(mean_squared_error(raw_values[-12:], predictions))
        print('%d) Test RMSE: %.3f' % (r+1, rmse))
        error_scores.append(rmse)
    return error_scores

# load dataset
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=
# experiment
repeats = 30
results = DataFrame()
lag = 1
neurons = 1
# vary training epochs
epochs = [50, 100, 500, 1000, 2000]
for e in epochs:
    results[str(e)] = experiment(repeats, series, e, lag, neurons)
# summarize results
print(results.describe())
# save boxplot
results.boxplot()
pyplot.savefig('boxplot_epochs.png')
```


Running the experiment prints the test set RMSE at the end of each experimental run.

At the end of all runs, a table of summary statistics is provided, one row for each statistic and one co[redacted]

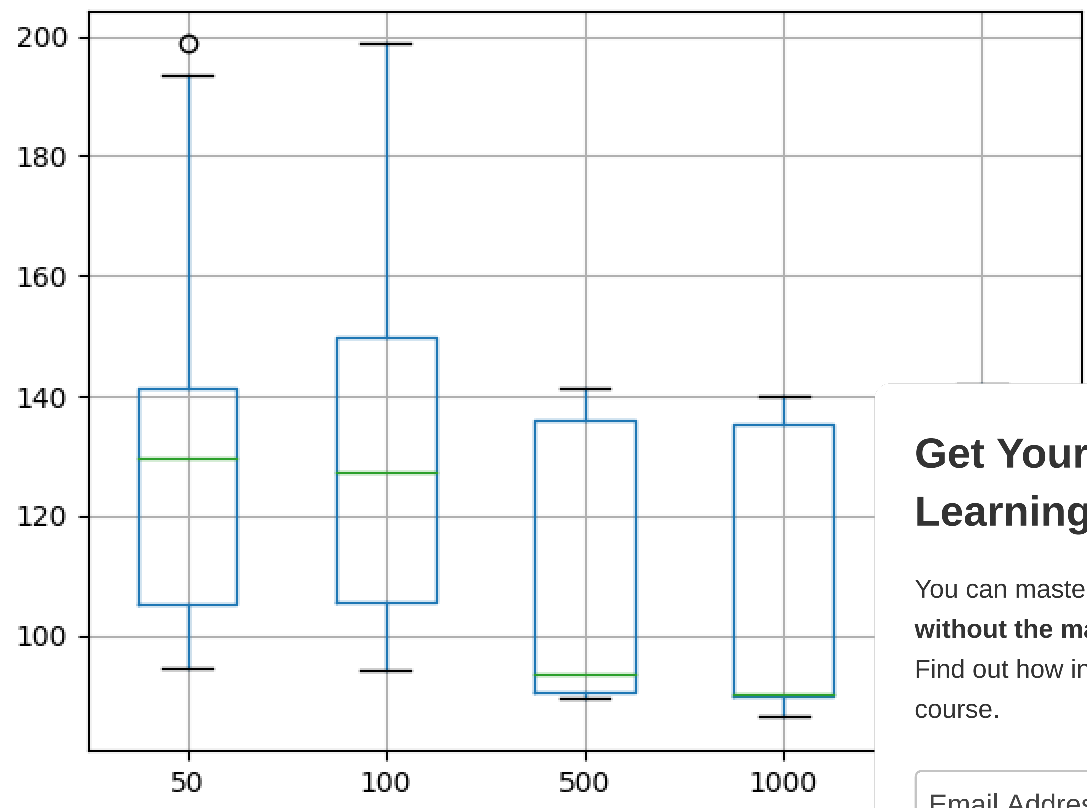
The summary statistics suggest that on average 1000 training epochs resulted in the better performance with a general decreasing trend in error with the increase of training epochs.

```
                50         100         500        1000        2000
count    30.000000   30.000000   30.000000   30.000000   30.000000
mean    129.660167  129.388944  111.444027  103.821703  107.500301
std      30.926344   28.499592   23.181317   22.138705   24.780781
min      94.598957   94.184903   89.506815   86.511801   86.452041
25%     105.198414  105.722736   90.679930   90.058655   86.457260
50%     129.705407  127.449491   93.508245   90.118331   90.074494
75%     141.420145  149.625816  136.157299  135.510850  135.741340
max     198.716220  198.704352  141.226816  139.994388  142.097747
```

A box and whisker plot of the distribution of test RMSE scores for each configuration was also create

The plot highlights that each configuration shows the same general spread in test RMSE scores (box d with the increase of training epochs.

The results confirm that the configured MLP trained for 1000 is a good starting point on this problem



Box and Whisker Plot of Varying Training Epochs for Time Series Forecasting on the

Another angle to consider with a network configuration is how it behaves over time as the model is being fit.

We can evaluate the model on the training and test datasets after each training epoch to get an idea as to if the configuration is overfitting or underfitting the problem.

We will use this diagnostic approach on the top result from each set of experiments. A total of 10 repeats of the configuration will be run and the train and test RMSE scores after each training epoch plotted on a line plot.

In this case, we will use this diagnostic on the MLP fit for 1000 epochs.

The complete diagnostic code listing is provided below.

As with the previous code listing, the code listing below will be used as the basis for all diagnostics in this tutorial and only the changes to this listing will be provided in subsequent sections.

```
from pandas import DataFrame
from pandas import Series
from pandas import concat
from pandas import read_csv
from pandas import datetime
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense
from math import sqrt
import matplotlib
# be able to save images on server
matplotlib.use('Agg')
from matplotlib import pyplot
import numpy

# date-time parsing function for loading the dataset
def parser(x):
    return datetime.strptime('190'+x, '%Y-%m')

# frame a sequence as a supervised learning problem
def timeseries_to_supervised(data, lag=1):
    df = DataFrame(data)
    columns = [df.shift(i) for i in range(1, lag+1)]
    columns.append(df)
    df = concat(columns, axis=1)
    df = df.drop(0)
    return df

# create a differenced series
def difference(dataset, interval=1):
    diff = list()
    for i in range(interval, len(dataset)):
        value = dataset[i] - dataset[i - interval]
        diff.append(value)
    return Series(diff)

# scale train and test data to [-1, 1]
def scale(train, test):
    # fit scaler
```

```
    scaler = MinMaxScaler(feature_range=(-1, 1))
    scaler = scaler.fit(train)
    # transform train
    train = train.reshape(train.shape[0], train.shape[1])
    train_scaled = scaler.transform(train)
    # transform test
    test = test.reshape(test.shape[0], test.shape[1])
    test_scaled = scaler.transform(test)
    return scaler, train_scaled, test_scaled

# inverse scaling for a forecasted value
def invert_scale(scaler, X, yhat):
    new_row = [x for x in X] + [yhat]
    array = numpy.array(new_row)
    array = array.reshape(1, len(array))
    inverted = scaler.inverse_transform(array)
    return inverted[0, -1]

# evaluate the model on a dataset, returns RMSE in transformed units
def evaluate(model, raw_data, scaled_dataset, scaler, offset, batch_size):
    # separate
    X, y = scaled_dataset[:,0:-1], scaled_dataset[:,-1]
    # forecast dataset
    output = model.predict(X, batch_size=batch_size)
    # invert data transforms on forecast
    predictions = list()
    for i in range(len(output)):
        yhat = output[i,0]
        # invert scaling
        yhat = invert_scale(scaler, X[i], yhat)
        # invert differencing
        yhat = yhat + raw_data[i]
        # store forecast
        predictions.append(yhat)
    # report performance
    rmse = sqrt(mean_squared_error(raw_data[1:], predictions))
    return rmse

# fit an MLP network to training data
def fit(train, test, raw, scaler, batch_size, nb_epoch, neurons):
    X, y = train[:, 0:-1], train[:, -1]
    # prepare model
    model = Sequential()
    model.add(Dense(neurons, activation='relu', input_dim=X.shape[1]))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')
    # fit model
```

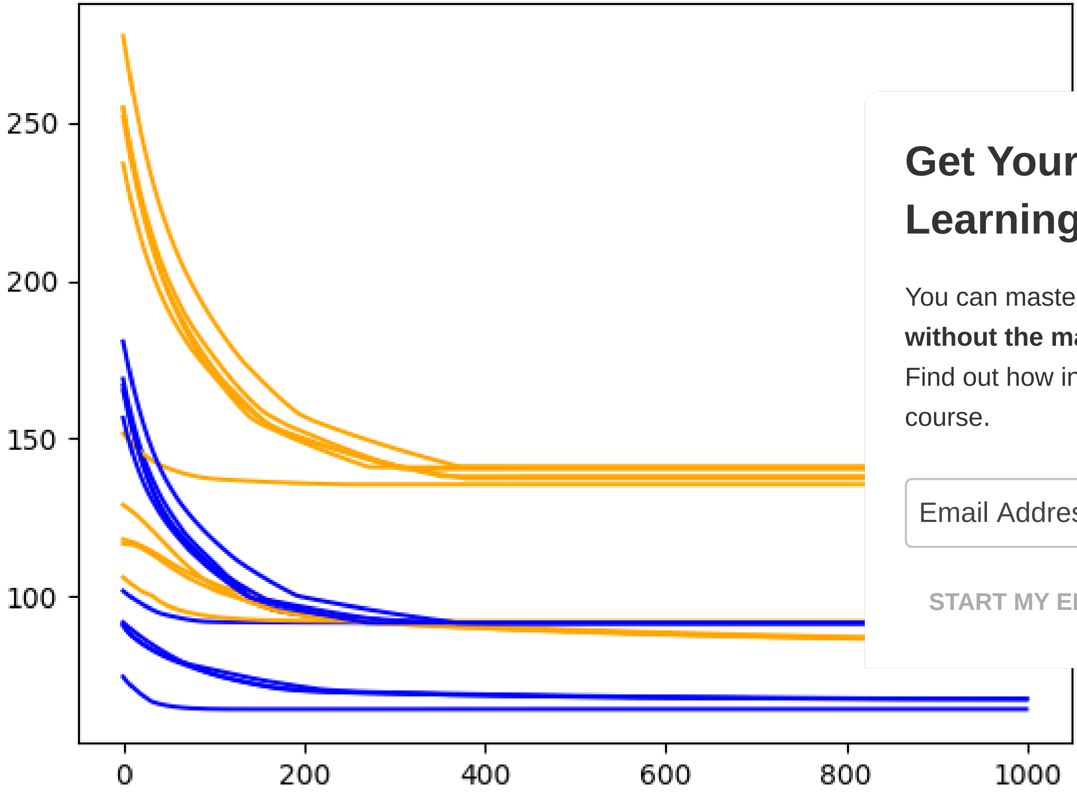
```
    train_rmse, test_rmse = list(), list()
    for i in range(nb_epoch):
        model.fit(X, y, epochs=1, batch_size=batch_size, verbose=0, shuffle=False)
        # evaluate model on train data
        raw_train = raw[-(len(train)+len(test)+1):-len(test)]
        train_rmse.append(evaluate(model, raw_train, train, scaler, 0, batch_size))
        # evaluate model on test data
        raw_test = raw[-(len(test)+1):]
        test_rmse.append(evaluate(model, raw_test, test, scaler, 0, batch_size))
    history = DataFrame()
    history['train'], history['test'] = train_rmse, test_rmse
    return history

# run diagnostic experiments
def run():
    # config
    repeats = 10
    n_batch = 4
    n_epochs = 1000
    n_neurons = 1
    n_lag = 1
    # load dataset
    series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, sque
    # transform data to be stationary
    raw_values = series.values
    diff_values = difference(raw_values, 1)
    # transform data to be supervised learning
    supervised = timeseries_to_supervised(diff_values, n_lag)
    supervised_values = supervised.values[n_lag:,:]
    # split data into train and test-sets
    train, test = supervised_values[0:-12], supervised_values[-12:]
    # transform the scale of the data
    scaler, train_scaled, test_scaled = scale(train, test)
    # fit and evaluate model
    train_trimmed = train_scaled[2:, :]
    # run diagnostic tests
    for i in range(repeats):
        history = fit(train_trimmed, test_scaled, raw_values, scaler, n_batch, n_epochs, n_neurons)
        pyplot.plot(history['train'], color='blue')
        pyplot.plot(history['test'], color='orange')
        print('%d) TrainRMSE=%f, TestRMSE=%f' % (i, history['train'].iloc[-1], history['test'].iloc[-1]))
    pyplot.savefig('diagnostic_epochs.png')

# entry point
run()
```


Running the diagnostic prints the final train and test RMSE for each run. More interesting is the final

The line plot shows the train RMSE (blue) and test RMSE (orange) after each training epoch.

In this case, the diagnostic plot shows little difference in train and test RMSE after about 400 training epochs. Both train and test performance level out on a near flat line.

This rapid leveling out suggests the model is reaching capacity and may benefit from more information in terms of lag observations or additional neurons.



Diagnostic Line Plot of Train and Test Performance of 1000 Epochs on the Shampoo Sales Dataset

Vary Hidden Layer Neurons

In this section, we will look at varying the number of neurons in the single hidden layer.

Increasing the number of neurons can increase the learning capacity of the network at the risk of overfitting the training data.

We will explore increasing the number of neurons from 1 to 5 and fit the network for 1000 epochs.

The differences in the experiment script are listed below.

```
# load dataset
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=True, date_parser=parser)
# experiment
repeats = 30
results = DataFrame()
lag = 1
epochs = 1000
# vary neurons
neurons = [1, 2, 3, 4, 5]
for n in neurons:
    results[str(n)] = experiment(repeats, series, epochs, lag, n)
# summarize results
print(results.describe())
# save boxplot
results.boxplot()
pyplot.savefig('boxplot_neurons.png')
```


Running the experiment prints summary statistics for each configuration.

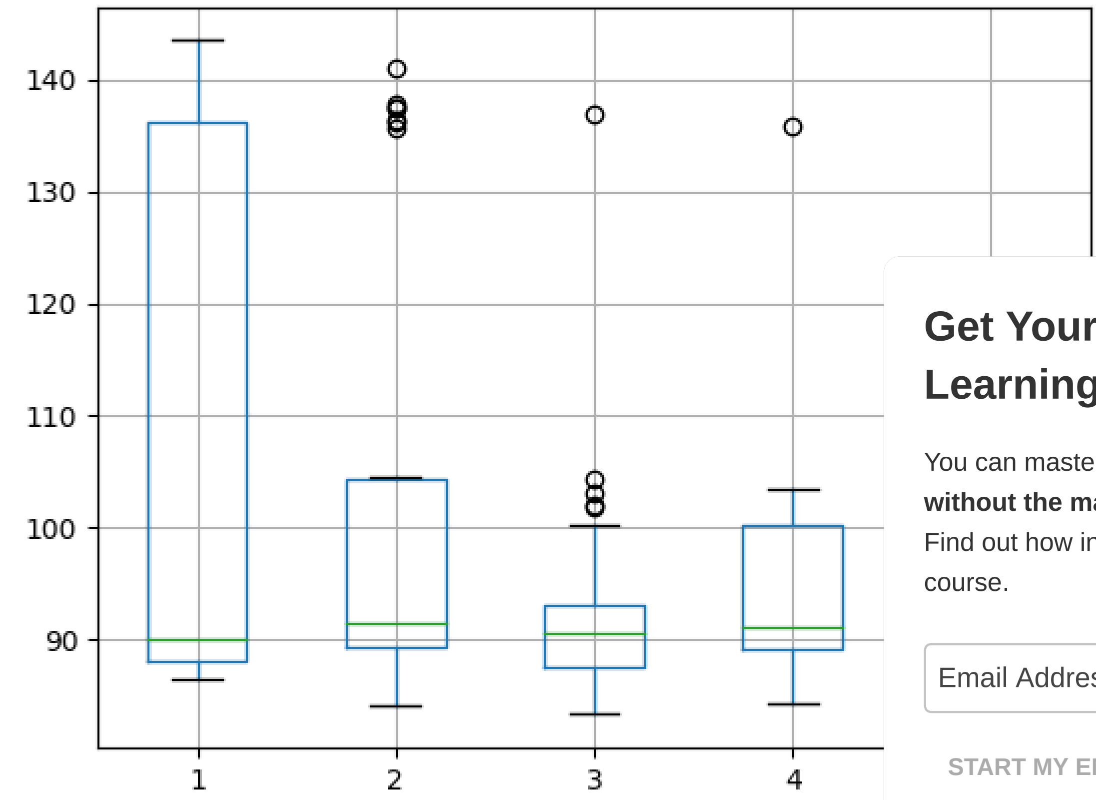
Looking at the average performance, it suggests a decrease of test RMSE with an increase in the nu

The best results appear to be with 3 neurons.

```
                1           2           3           4           5
count   30.000000   30.000000   30.000000   30.000000   30.000000
mean   105.107026  102.836520   92.675912   94.889952   96.577617
std     23.130824   20.102353   10.266732    9.751318    6.421356
min     86.565630   84.199871   83.388967   84.385293   87.208454
25%     88.035396   89.386670   87.643954   89.154866   89.961809
50%     90.084895   91.488484   90.670565   91.204303   96.717739
75%    136.145248  104.416518   93.117926  100.228730  101.969331
max    143.428154  140.923087  136.883946  135.891663  106.797563
```

A box and whisker plot is also created to summarize and compare the distributions of results.

The plot confirms the suggestion of 3 neurons performing well compared to the other configurations [redacted] is also smaller. This may indicate a more stable configuration.



Box and Whisker Plot of Varying Hidden Neurons for Time Series Forecasting on the Shampoo Sales Dataset

Again, we can dive a little deeper by reviewing diagnostics of the chosen configuration of 3 neurons fit for 1000 epochs.

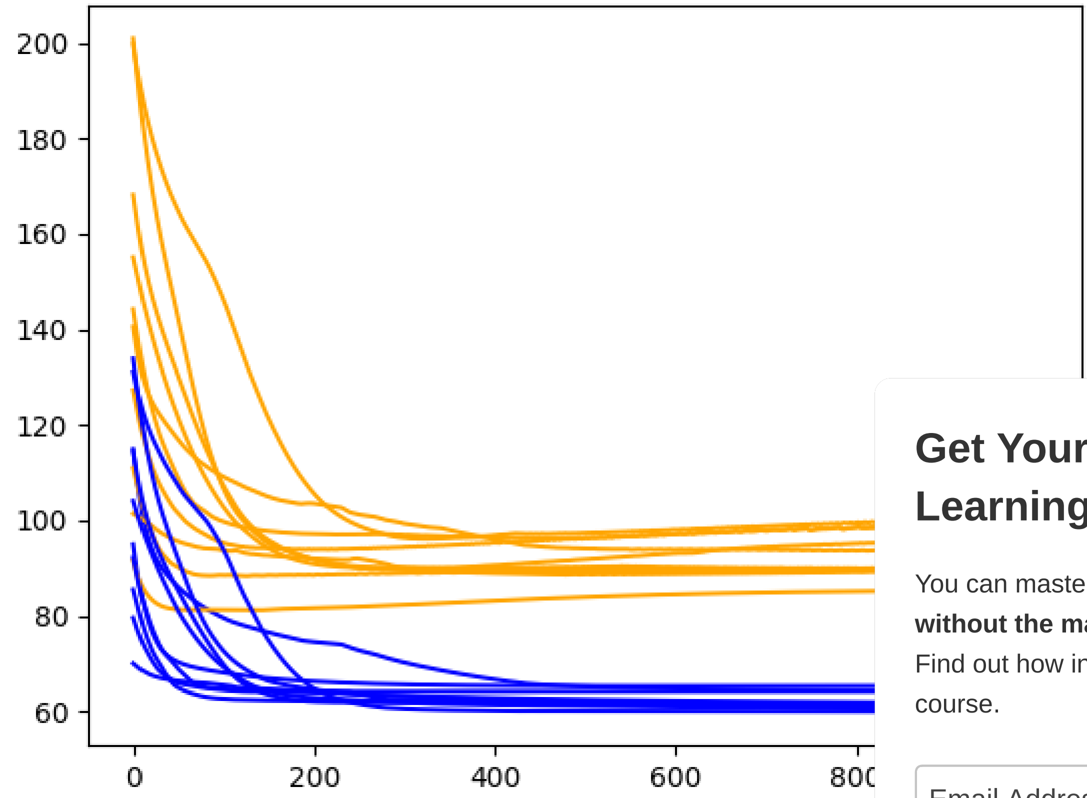
The changes to the diagnostic script are limited to the *run()* function and listed below.

```
# run diagnostic experiments
def run():
```

```
    # config
    repeats = 10
    n_batch = 4
    n_epochs = 1000
    n_neurons = 3
    n_lag = 1
    # load dataset
    series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=True, date_parser=parser)
    # transform data to be stationary
    raw_values = series.values
    diff_values = difference(raw_values, 1)
    # transform data to be supervised learning
    supervised = timeseries_to_supervised(diff_values, n_lag)
    supervised_values = supervised.values[n_lag:,:]
    # split data into train and test-sets
    train, test = supervised_values[0:-12], supervised_values[-12:]
    # transform the scale of the data
    scaler, train_scaled, test_scaled = scale(train, test)
    # fit and evaluate model
    train_trimmed = train_scaled[2:, :]
    # run diagnostic tests
    for i in range(repeats):
        history = fit(train_trimmed, test_scaled, raw_values, scaler, n_batch, n_epochs,
        pyplot.plot(history['train'], color='blue')
        pyplot.plot(history['test'], color='orange')
        print('%d) TrainRMSE=%f, TestRMSE=%f' % (i, history['train'].iloc[-1], history['
    pyplot.savefig('diagnostic_neurons.png')
```

Running the diagnostic script provides a line plot of train and test RMSE for each training epoch.

The diagnostics suggest a flattening out of model skill, perhaps around 400 epochs. The plot also su ere is a slight increase in test RMSE over the last 500 training epochs, but not a strong increase in traini



Diagnostic Line Plot of Train and Test Performance of 3 Hidden Neurons on the S

Vary Hidden Layer Neurons with Lag

In this section, we will look at increasing the lag observations as input, whilst at the same time increasing the capacity of the network.

Increased lag observations will automatically scale the number of input neurons. For example, 3 lag observations as input will result in 3 input neurons.

The added input will require additional capacity in the network. As such, we will also scale the number of neurons in the one hidden layer with the number of lag observations used as input.

We will use odd numbers of lag observations as input from 1, 3, 5, and 7 and use the same number [redacted]

The change to the number of inputs affects the total number of training patterns during the conversion of the time series data to a supervised learning problem. As such, the batch size was reduced from 4 to 2 for all experiments in this section.

A total of 1000 training epochs are used in each experimental run.

The changes from the base experiment script are limited to the *experiment()* function and the running of the experiment, listed below.

```
# run a repeated experiment
def experiment(repeats, series, epochs, lag, neurons):
    # transform data to be stationary
    raw_values = series.values
    diff_values = difference(raw_values, 1)
    # transform data to be supervised learning
    supervised = timeseries_to_supervised(diff_values, lag)
    supervised_values = supervised.values[lag:,:]
    # split data into train and test-sets
    train, test = supervised_values[0:-12], supervised_values[-12:]
    # transform the scale of the data
    scaler, train_scaled, test_scaled = scale(train, test)
    # run experiment
    error_scores = list()
    for r in range(repeats):
        # fit the model
        batch_size = 2
        model = fit_model(train_scaled, batch_size, epochs, neurons)
        # forecast test dataset
        test_reshaped = test_scaled[:,0:-1]
        output = model.predict(test_reshaped, batch_size=batch_size)
        predictions = list()
        for i in range(len(output)):
            yhat = output[i,0]
            X = test_scaled[i, 0:-1]
            # invert scaling
            yhat = invert_scale(scaler, X, yhat)
            # invert differencing
            yhat = inverse_difference(raw_values, yhat, len(test_scaled)+1-i)
            # store forecast
            predictions.append(yhat)
        # report performance
        rmse = sqrt(mean_squared_error(raw_values[-12:], predictions))
        print('%d) Test RMSE: %.3f' % (r+1, rmse))
        error_scores.append(rmse)
```

```
    return error_scores

# load dataset
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=True, date_parser=parser)
# experiment
repeats = 30
results = DataFrame()
epochs = 1000
# vary neurons
neurons = [1, 3, 5, 7]
for n in neurons:
    results[str(n)] = experiment(repeats, series, epochs, n, n)
# summarize results
print(results.describe())
# save boxplot
results.boxplot()
pyplot.savefig('boxplot_neurons_lag.png')
```

Running the experiment summarizes the results using descriptive statistics for each configuration.

The results suggest that all increases in lag input variables with increases with hidden neurons decre

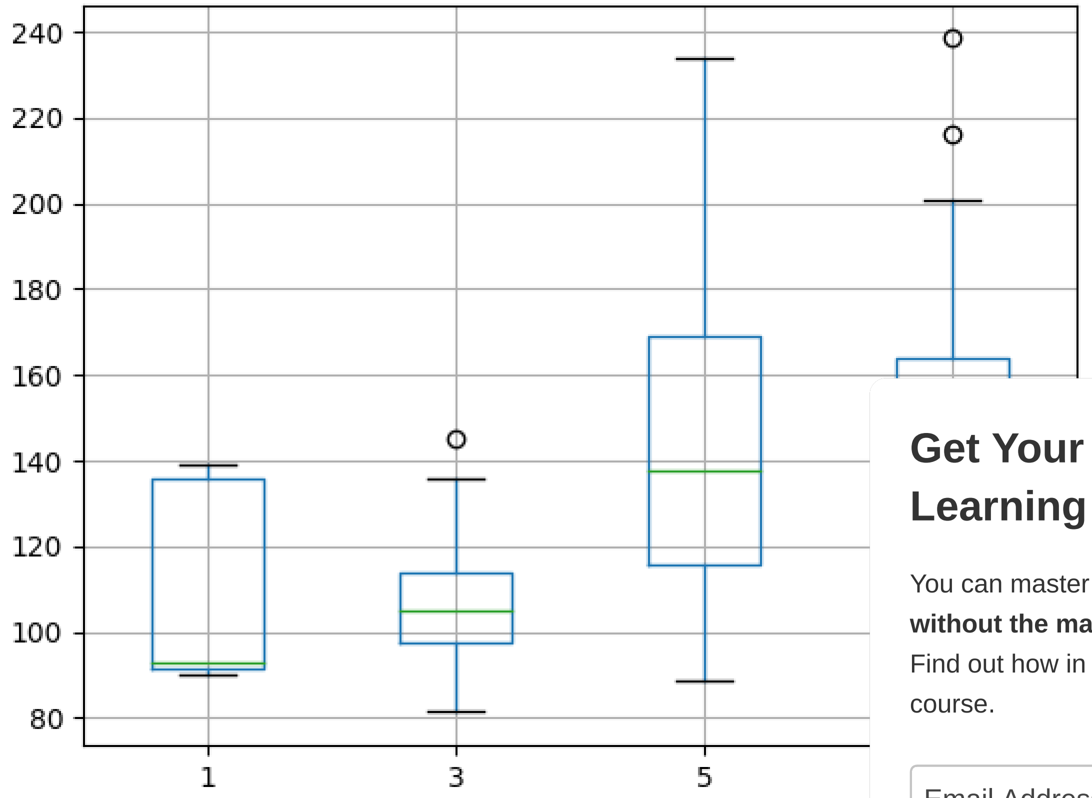
Of note is the 1 neuron and 1 input configuration, which compared to the results from the previous s
deviation.

It is possible that the decrease in performance is related to the smaller batch size and that the result
tease this out.

```
                1           3           5           7
count   30.000000   30.000000   30.000000   30.000000
mean   105.465038  109.447044  158.894730  147.024776
std     20.827644   15.312300   43.177520   22.717514
min     89.909627   77.426294   88.515319   95.801699
25%     92.187690  102.233491  125.008917  132.335683
50%     92.587411  109.506480  166.438582  145.078842
75%    135.386125  118.635143  189.457325  166.329000
max    139.941789  144.700754  232.962778  186.185471
```

A box and whisker plot of the distribution of results was also created allowing configurations to be compared.

Interestingly, the use of 3 neurons and 3 input variables shows a tighter spread compared to the other configurations. This is similar to the observation
from 3 neurons and 1 input variable seen in the previous section.



Box and Whisker Plot of Varying Lag Features and Hidden Neurons for Time Series Fore
Dataset

We can also use diagnostics to tease out how the dynamics of the model might have changed while fitting the model.

The results for 3-lags/3-neurons are interesting and we will investigate them further.

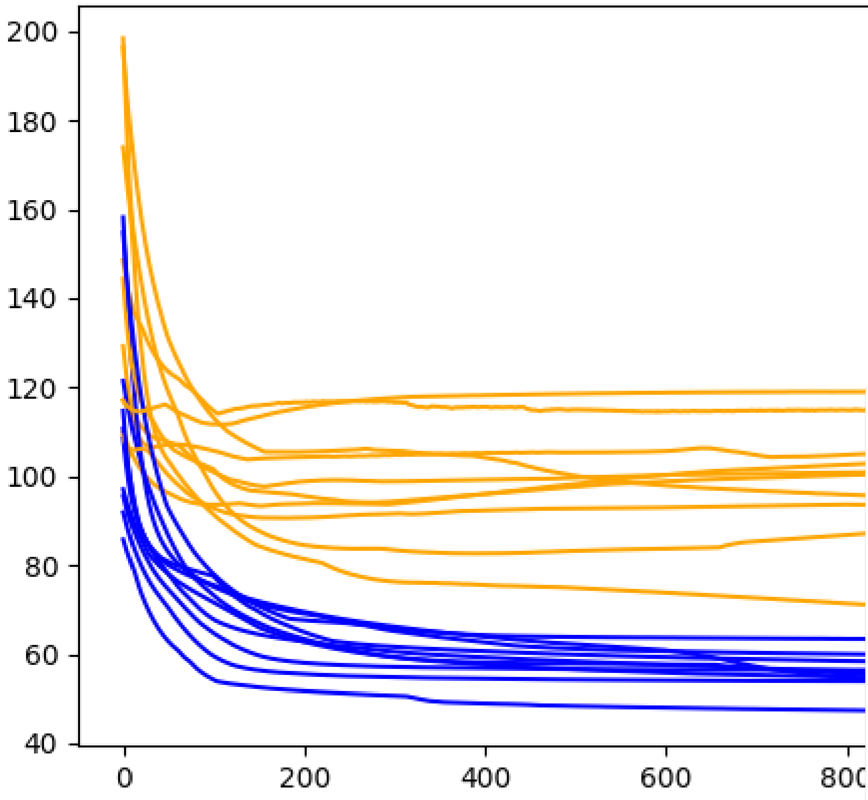
The changes to the diagnostic script are confined to the *run()* function.

```
# run diagnostic experiments
def run():
    # config
```

```
	repeats = 10
	n_batch = 2
	n_epochs = 1000
	n_neurons = 3
	n_lag = 3
	# load dataset
	series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0, squeeze=True, date_parser=parser)
	# transform data to be stationary
	raw_values = series.values
	diff_values = difference(raw_values, 1)
	# transform data to be supervised learning
	supervised = timeseries_to_supervised(diff_values, n_lag)
	supervised_values = supervised.values[n_lag:,:]
	# split data into train and test-sets
	train, test = supervised_values[0:-12], supervised_values[-12:]
	# transform the scale of the data
	scaler, train_scaled, test_scaled = scale(train, test)
	# fit and evaluate model
	train_trimmed = train_scaled[2:, :]
	# run diagnostic tests
	for i in range(repeats):
		history = fit(train_trimmed, test_scaled, raw_values, scaler, n_batch, n_epochs,
		pyplot.plot(history['train'], color='blue')
		pyplot.plot(history['test'], color='orange')
		print('%d) TrainRMSE=%f, TestRMSE=%f' % (i, history['train'].iloc[-1], history['
	pyplot.savefig('diagnostic_neurons_lag.png')
```

Running the diagnostics script creates a line plot showing the train and test RMSE after each trainin

The results suggest good learning during the first 500 epochs and perhaps overfitting in the remainir ing trend and the train RMSE showing a decreasing trend.



Diagnostic Line Plot of Train and Test Performance of 3 Hidden Neurons and Lag Feat
Dataset

Review of Results

We have covered a lot of ground in this tutorial. Let's review.

- **Epochs**. We looked at how model skill varied with the number of training epochs and found that 1000 might be a good starting point.
- **Neurons**. We looked at varying the number of neurons in the hidden layer and found that 3 neurons might be a good configuration.

- **Lag Inputs**. We looked at varying the number of lag observations as inputs whilst at the same [redacted] layer and found that results generally got worse, but again, 3 neurons in the hidden layer shows interest. Poor results may have been related to the change of batch size from 4 to 2 compared to other experiments.

The results suggest using a 1 lag input, 3 neurons in the hidden layer, and fit for 1000 epochs as a first-cut model configuration.

This can be improved upon in many ways; the next section lists some ideas.

Extensions

This section lists extensions and follow-up experiments you might like to explore.

- **Shuffle vs No Shuffle**. No shuffling was used, which is abnormal. Develop an experiment to co when fitting the model for time series forecasting.
- **Normalization Method**. Data was rescaled to -1 to 1, typical for a tanh activation function, not u rescaling, such as 0-1 normalization and standardization and the impact on model performance
- **Multiple Layers**. Explore the use of multiple hidden layers to add network capacity to learn mor
- **Feature Engineering**. Explore the use of additional features, such as an error time series and e n.

Also, check out the post:

- How To Improve Deep Learning Performance

Did you try any of these extensions?
Post your results in the comments below.

Summary

In this tutorial, you discovered how to use systematic experiments to explore the configuration of a multilayer perceptron for time series forecasting and develop a first-cut model.

Specifically, you learned:

- How to develop a robust test harness for evaluating MLP models for time series forecasting.

- How to systematically evaluate training epochs, hidden layer neurons, and lag inputs.
- How to use diagnostics to help interpret results and suggest follow-up experiments.

Do you have any questions about this tutorial?
Ask your questions in the comments below and I will do my best to answer.

About Jason Brownlee

Dr. Jason Brownlee is a husband, proud father, academic researcher, author, professional devel [redacted]
to helping developers get started and get good at applied machine learning. Learn more.
View all posts by Jason Brownlee →

‹ Instability of Online Learning for Stateful LSTM for Time Series Forecasting

How to Use Dropout with LSTM Networks for Time Series Forecasting ›

20 Responses to *Exploratory Configuration of a Multilayer Perceptron Netw*

elina April 26, 2017 at 11:00 pm #

what is mean by transforming observations to have specific scale? can u more elaborate it with

Jason Brownlee April 27, 2017 at 8:40 am #

Yes, normalize each column to the range 0-1.

Kunpeng Zhang April 28, 2017 at 3:12 am #

REPLY

Hi Jason,
I have a question regarding the prediction result evaluation.
In my code, I get MSE MAE MAPE calculated by keras as follows:
model.compile(loss='mse', optimizer=optimizer, metrics=['mae', 'mape'])
…
Epoch 150/150

0s – loss: 0.0038 – mean_absolute_error: 0.0455 – mean_absolute_percentage_error: 164095.5176
mse=0.003538, mae=0.043654, mape=58238.251235
———————————————————————————-

But when I compute these values by the code:
mse = mean_squared_error(test_Y, predicted_output)
rmse = math.sqrt(mse)
mae = mean_absolute_error(test_Y, predicted_output)
mape = np.mean(np.abs(np.divide(np.subtract(test_Y, predicted_output), test_Y))) * 100
RMSE: 14.992
MAE: 10.462
MAPE: 2.208

The two results are very different from each other. What happened?

Jason Brownlee April 28, 2017 at 7:53 am #

That is interesting. I'm not sure what is going on here.

I would trust the manual results. I have not had any issues like this myself, my epoch scores always

Consider preparing a small self-contained example and posting it as a bug to the Keras project:
https://github.com/fchollet/keras

Kunpeng Zhang April 29, 2017 at 11:46 pm #

I give it a try on your example.
http://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

Change
model.compile(loss='mean_squared_error', optimizer='adam')
to
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mae', 'mape'])

Got the same result.
0s – loss: 0.0019 – mean_absolute_error: 0.0343 – mean_absolute_percentage_error: 620693.8651
Epoch 96/100
0s – loss: 0.0020 – mean_absolute_error: 0.0351 – mean_absolute_percentage_error: 326543.9207
Epoch 97/100
0s – loss: 0.0019 – mean_absolute_error: 0.0345 – mean_absolute_percentage_error: 488762.9108
Epoch 98/100
0s – loss: 0.0019 – mean_absolute_error: 0.0345 – mean_absolute_percentage_error: 514091.1566
Epoch 99/100
0s – loss: 0.0019 – mean_absolute_error: 0.0345 – mean_absolute_percentage_error: 531419.0410
Epoch 100/100
0s – loss: 0.0019 – mean_absolute_error: 0.0341 – mean_absolute_percentage_error: 454424.3737
Train Score: 22.34 RMSE
Test Score: 45.66 RMSE

Hans April 28, 2017 at 6:12 pm #

What is the most robust sign for overfitting?

Jason Brownlee April 29, 2017 at 7:22 am #

Skill on the test set is worse than the training set.

Hans April 30, 2017 at 3:50 pm #

REPLY

Is there a list of overfitting signs available?

Alternative:

If I want to prepare such a list, how should it look like?

1. Skill on the test set is worse than the training set.
2. ?
3. ?
4. ?
5. ?

Hans April 30, 2017 at 3:51 pm # REPLY

Is there a function available for our code, which ‘alerts’ overfitting?

Hans April 30, 2017 at 4:00 pm #

I monitor every result and multiple parameters of your examples in a Sqlite data

Therefore I could easelly tagging overfitting.

For example if a test result is worse then the training set.

Could there be more relations of parameters and results to monitor for overfitting in gene

Jason Brownlee May 1, 2017 at 5:55 am #

It is a trend you are seeking for overfitting, not necessarily one result being worse.
https://en.wikipedia.org/wiki/Overfitting

Jason Brownlee May 1, 2017 at 5:54 am #

No.

Jason Brownlee May 1, 2017 at 5:54 am # REPLY

Nope. The first is all you need.

Instead, you develop a list of 100s of ways to address overfitting/early convergence.

Hans May 1, 2017 at 9:21 am #

A)
Does "Skill on the test set is worse than the training set"
mean trainRmse is less then testRmse?

In my baseline-test my trainRmse is always zero and the testRmse is always higher? Is

B)
Could we say that a higher variance of test values is an indicator for overfitting?

Jason Brownlee May 2, 2017 at 5:56 am #

This post might make things clearer for you Hans:
http://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-al

Hans April 30, 2017 at 4:52 pm # REPLY

Could we assume that the best performing epoches, neurons and lag inputs found with this MLP setup are also the best performing ones in a LSTM model or any other model?

[redacted]

Jason Brownlee May 1, 2017 at 5:56 am #

No, generally findings are not transferable to other problems or other algorithms.

Hans May 1, 2017 at 8:59 pm #

REPLY

Is it possible to save a trained model on HD, to predict unseen data later, in a shorter time?

Magnus May 12, 2017 at 11:22 pm #

REPLY

Would it be possible to achieve the same study using GridSearchCV, like you did in another pos

Jason Brownlee May 13, 2017 at 6:15 am #

No, we must use walk-forward validation to evaluate time series models correctly:
http://machinelearningmastery.com/backtest-machine-learning-models-time-series-forecasting/

Leave a Reply

Name (required)

Email (will not be published) (required)

Website

SUBMIT COMMENT

Welcome to Machine Learning Mastery

Hi, I'm Dr. Jason Brownlee.
My goal is to make practitioners like YOU awesome at applied machine learning.

Read More

Finally Get Started With Deep Learning

Sick of the fancy math and need for super computers?
Looking for step-by-step tutorials?

POPULAR

Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras
JULY 21, 2016

Your First Machine Learning Project in Python Step-By-Step
JUNE 10, 2016

Develop Your First Neural Network in Python With Keras Step-By-Step
MAY 24, 2016

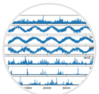
Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras
JULY 26, 2016

How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda
MARCH 13, 2017

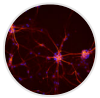
Time Series Forecasting with the Long Short-Term Memory Network in Python
APRIL 7, 2017

Multi-Class Classification Tutorial with the Keras Deep Learning Library
JUNE 2, 2016

Regression Tutorial with the Keras Deep Learning Library in Python
JUNE 9, 2016



Multivariate Time Series Forecasting with LSTMs in Keras
AUGUST 14, 2017



How to Implement the Backpropagation Algorithm From Scratch In Python
NOVEMBER 7, 2016

Privacy | Contact | About