**Bias Weight Regularization**

Weight regularization can be applied to the bias connection within the LSTM nodes.

In Keras, this is specified with a *bias\_regularizer* argument when creating an LSTM layer. The regularizer is defined as an instance of the one of the L1, L2, or L1L2 classes.

More details here:

* [Keras Usage of Regularizers](https://keras.io/regularizers/)

In this experiment, we will compare L1, L2, and L1L2 with a default value of 0.01 against the baseline model. We can specify all configurations using the L1L2 class, as follows:

* L1L2(0.0, 0.0) [e.g. baseline]
* L1L2(0.01, 0.0) [e.g. L1]
* L1L2(0.0, 0.01) [e.g. L2]
* L1L2(0.01, 0.01) [e.g. L1L2 or elasticnet]

Below lists the updated *fit\_lstm()*, *experiment()*, and *run()* functions for using bias weight regularization with LSTMs.

## Input Weight Regularization

We can also apply regularization to input connections on each LSTM unit.

In Keras, this is achieved by setting the kernel\_regularizer argument to a regularizer class.

We will test the same regularizer configurations as were used in the previous section, specifically:

* L1L2(0.0, 0.0) [e.g. baseline]
* L1L2(0.01, 0.0) [e.g. L1]
* L1L2(0.0, 0.01) [e.g. L2]
* L1L2(0.01, 0.01) [e.g. L1L2 or elasticnet]

Below lists the updated fit\_lstm(), experiment(), and run() functions for using bias weight regularization with LSTMs.

**Input Weight Regularization**

We can also apply regularization to input connections on each LSTM unit.

In Keras, this is achieved by setting the *kernel\_regularizer* argument to a regularizer class.

We will test the same regularizer configurations as were used in the previous section, specifically:

* L1L2(0.0, 0.0) [e.g. baseline]
* L1L2(0.01, 0.0) [e.g. L1]
* L1L2(0.0, 0.01) [e.g. L2]
* L1L2(0.01, 0.01) [e.g. L1L2 or elasticnet]

Below lists the updated *fit\_lstm()*, *experiment()*, and *run()* functions for using bias weight regularization with LSTMs.

The results clearly show a downward trend in RMSE over the training epochs for almost all of the experimental runs.

This is a good sign, as it shows the model is learning the problem and has some predictive skill. In fact, all of the final test scores are below the error of a simple persistence model (naive forecast) that achieves an RMSE of 136.761 on this problem.

The results suggest that more training epochs will result in a more skillful model.

Let’s try doubling the number of epochs from 500 to 1000.

### Diagnostic of 1000 Epochs

In this section, we use the same experimental setup and fit the model over 1000 training epochs.

Specifically, the n\_epochs parameter is set to 1000 in the run() function.

We can see that the downward trend of model error does continue and appears to slow.

The lines for the train and test cases become more horizontal, but still generally show a downward trend, although at a lower rate of change. Some examples of test error show a possible inflection point around 600 epochs and may show a rising trend.

It is worth extending the epochs further. We are interested in the average performance continuing to improve on the test set and this may continue.

Let’s try doubling the number of epochs from 1000 to 2000.

### Diagnostic of 2000 Epochs

In this section, we use the same experimental setup and fit the model over 2000 training epochs.

Specifically, the n\_epochs parameter is set to 2000 in the run() function.

As one might have guessed, the downward trend in error continues over the additional 1000 epochs on both the train and test datasets.

Of note, about half of the cases continue to decrease in error all the way to the end of the run, whereas the rest show signs of an increasing trend.

The increasing trend is a sign of overfitting. This is when the model overfits the training dataset at the cost of worse performance on the test dataset. It is exemplified by continued improvements on the training dataset and improvements followed by an inflection point and worsting skill in the test dataset. A little less than half of the runs show the beginnings of this type of pattern on the test dataset.

Nevertheless, the final epoch results on the test dataset are very good. If there is a chance we can see further gains by even longer training, we must explore it.

Let’s try doubling the number of epochs from 2000 to 4000.

### Diagnostic of 4000 Epochs

In this section, we use the same experimental setup and fit the model over 4000 training epochs.

A similar pattern continues.

There is a general trend of improving performance, even over the 4000 epochs. There is one case of severe overfitting where test error rises sharply.

Again, most runs end with a “good” (better than persistence) final test error.

### Summary of Results

The diagnostic runs above are helpful to explore the dynamical behavior of the model, but fall short of an objective and comparable mean performance.

We can address this by repeating the same experiments and calculating and comparing summary statistics for each configuration. In this case, 30 runs were completed of the epoch values 500, 1000, 2000, 4000, and 6000.

The idea is to compare the configurations using summary statistics over a larger number of runs and see exactly which of the configurations might perform better on average.

The complete code example is listed below.