



[Click to Take the FREE Deep Learning Performance Crash-Course](#)



# How to Reduce Generalization Error With Activity Regularization in Keras

by **Jason Brownlee** on November 30, 2018 in **Deep Learning Performance**

Tweet

Share

Share

Last Updated on October 3, 2019

Activity regularization provides an approach to encourage a neural network to learn sparse features or internal representations of raw observations.

It is common to seek sparse learned representations in autoencoders, called sparse autoencoders, and in encoder-decoder models, although the approach can also be used generally to [reduce overfitting and improve a model's ability to generalize](#) to new observations.

In this tutorial, you will discover the Keras API for adding activity regularization to deep learning neural network models.

After completing this tutorial, you will know:

- How to create vector norm regularizers using the Keras API.
- How to add activity regularization to MLP, CNN, and RNN layers using the Keras API.
- How to reduce overfitting by adding activity regularization to an existing model.

Discover how to train faster, reduce overfitting, and make better predictions with deep learning models in [my new book](#), with 26 step-by-step tutorials and full source code.

Let's get started.

- **Updated Oct/2019:** Updated for Keras 2.3 and TensorFlow 2.0.



How to Reduce Generalization Error in Deep Neural Networks With Activity Regularization in Keras  
Photo by [Johan Neven](#), some rights reserved.

## Tutorial Overview

This tutorial is divided into three parts; they are:

1. Activity Regularization in Keras
2. Activity Regularization on Layers
3. Activity Regularization Case Study

## Activity Regularization in Keras

Keras supports activity regularization.

There are three different regularization techniques supported, each provided as a class in the *keras.regularizers* module:

- **l1**: Activity is calculated as the sum of absolute values.
- **l2**: Activity is calculated as the sum of the squared values.
- **l1\_l2**: Activity is calculated as the sum of absolute and sum of the squared values.

Each of the *l1* and *l2* regularizers takes a single hyperparameter that controls the amount that each activity contributes to the sum. The *l1\_l2* regularizer takes two hyperparameters, one for each of the *l1* and *l2* methods.

The regularizer class must be imported and then instantiated; for example:

```
1 # import regularizer
```

```
2 from keras.regularizers import l1
3 # instantiate regularizer
4 reg = l1(0.001)
```

## Activity Regularization on Layers

Activity regularization is specified on a layer in Keras.

This can be achieved by setting the *activity\_regularizer* argument on the layer to an instantiated and configured regularizer class.

The regularizer is applied to the output of the layer, but you have control over what the “*output*” of the layer actually means. Specifically, you have flexibility as to whether the layer output means that the regularization is applied before or after the ‘*activation*’ function.

For example, you can specify the function and the regularization on the layer, in which case activation regularization is applied to the output of the activation function, in this case, [rectified linear activation function](#) or ReLU.

```
...
model.add(Dense(32, activation='relu', activity_regularizer=l1(0.001)))
...
```

Alternately, you can specify a linear activation function (the default, that does not perform any transform) which means that the activation regularization is applied on the raw outputs, then, the activation function can be added as a subsequent layer.

```
1 ...
2 model.add(Dense(32, activation='linear', activity_regularizer=l1(0.001)))
3 model.add(Activation('relu'))
4 ...
```

The latter is probably the preferred usage of activation regularization as described in “[Deep Sparse Rectifier Neural Networks](#)” in order to allow the model to learn to take activations to a true zero value in conjunction with the rectified linear activation function. Nevertheless, the two possible uses of activation regularization may be explored in order to discover what works best for your specific model and dataset.

Let’s take a look at how activity regularization can be used with some common layer types.

## MLP Activity Regularization

The example below sets l1 norm activity regularization on a Dense fully connected layer.

```
1 # example of l1 norm on activity from a dense layer
2 from keras.layers import Dense
3 from keras.regularizers import l1
4 ...
5 model.add(Dense(32, activity_regularizer=l1(0.001)))
6 ...
```

## CNN Activity Regularization

The example below sets l1 norm activity regularization on a Conv2D convolutional layer.

```
1 # example of l1 norm on activity from a cnn layer
2 from keras.layers import Conv2D
3 from keras.regularizers import l1
4 ...
5 model.add(Conv2D(32, (3,3), activity_regularizer=l1(0.001)))
6 ...
```

## RNN Activity Regularization

The example below sets l1 norm activity regularization on an LSTM recurrent layer.

```
1 # example of l1 norm on activity from an lstm layer
2 from keras.layers import LSTM
3 from keras.regularizers import l1
4 ...
5 model.add(LSTM(32, activity_regularizer=l1(0.001)))
6 ...
```

Now that we know how to use the activity regularization API, let's look at a worked example.

---

## Want Better Results with Deep Learning?

Take my free 7-day email crash course now (with sample code).

Click to sign-up and also get a free PDF Ebook version of the course.

[Download Your FREE Mini-Course](#)

---

## Activity Regularization Case Study

In this section, we will demonstrate how to use activity regularization to reduce overfitting of an MLP on a simple binary classification problem.

Although activity regularization is most often used to encourage sparse learned representations in autoencoder and encoder-decoder models, it can also be used directly within normal neural networks to achieve the same effect and improve the generalization of the model.

This example provides a template for applying activity regularization to your own neural network for classification and regression problems.

## Binary Classification Problem

We will use a standard binary classification problem that defines two two-dimensional concentric circles of observations, one circle for each class.

Each observation has two input variables with the same scale and a class output value of either 0 or 1. This dataset is called the “*circles*” dataset because of the shape of the observations in each class when plotted.



We can use the `make_circles()` function to generate observations from this problem. We will add noise to the data and seed the random number generator so that the same samples are generated each time the code is run.

```
1 # generate 2d classification dataset
2 X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
```

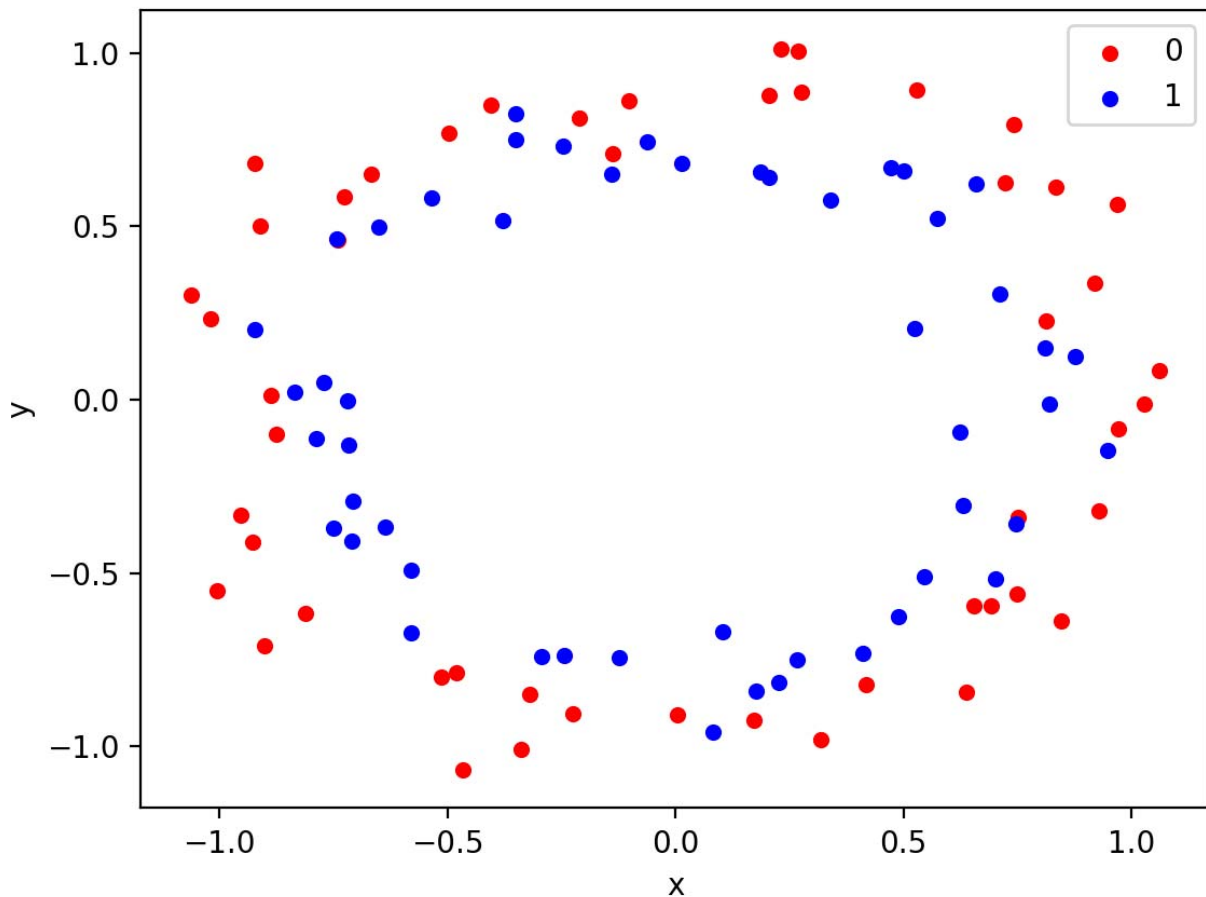
We can plot the dataset where the two variables are taken as x and y coordinates on a graph and the class value is taken as the color of the observation.

The complete example of generating the dataset and plotting it is listed below.

```
1 # generate two circles dataset
2 from sklearn.datasets import make_circles
3 from matplotlib import pyplot
4 from pandas import DataFrame
5 # generate 2d classification dataset
6 X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
7 # scatter plot, dots colored by class value
8 df = DataFrame(dict(x=X[:,0], y=X[:,1], label=y))
9 colors = {0:'red', 1:'blue'}
10 fig, ax = pyplot.subplots()
11 grouped = df.groupby('label')
12 for key, group in grouped:
13     group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])
14 pyplot.show()
```

Running the example creates a scatter plot showing the concentric circles shape of the observations in each class.

We can see the noise in the dispersal of the points making the circles less obvious.



Scatter Plot of Circles Dataset with Color Showing the Class Value of Each Sample

This is a good test problem because the classes cannot be separated by a line, e.g. are not linearly separable, requiring a nonlinear method such as a neural network to address.

We have only generated 100 samples, which is small for a neural network, providing the opportunity to overfit the training dataset and have higher error on the test dataset: a good case for using regularization.

Further, the samples have noise, giving the model an opportunity to learn aspects of the samples that don't generalize.

## Overfit Multilayer Perceptron

We can develop an MLP model to address this binary classification problem.

The model will have one hidden layer with more nodes that may be required to solve this problem, providing an opportunity to overfit. We will also train the model for longer than is required to ensure the model overfits.

Before we define the model, we will split the dataset into train and test sets, using 30 examples to train the model and 70 to evaluate the fit model's performance.

```
1 # generate 2d classification dataset
2 X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
3 # split into train and test
```

```

4 n_train = 30
5 trainX, testX = X[:n_train, :], X[n_train:, :]
6 trainy, testy = y[:n_train], y[n_train:]

```

Next, we can define the model.

The hidden layer uses 500 nodes and the rectified linear activation function. A sigmoid activation function is used in the output layer in order to predict class values of 0 or 1.

The model is optimized using the binary cross entropy loss function, suitable for binary classification problems and the efficient Adam version of gradient descent.

```

1 # define model
2 model = Sequential()
3 model.add(Dense(500, input_dim=2, activation='relu'))
4 model.add(Dense(1, activation='sigmoid'))
5 model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

```

The defined model is then fit on the training data for 4,000 epochs and the default batch size of 32.

We will also use the test dataset as a validation dataset.

```

1 # fit model
2 history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)

```

We can evaluate the performance of the model on the test dataset and report the result.

```

1 # evaluate the model
2 _, train_acc = model.evaluate(trainX, trainy, verbose=0)
3 _, test_acc = model.evaluate(testX, testy, verbose=0)
4 print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))

```

Finally, we will plot the performance of the model on both the train and test set each epoch.

If the model does indeed overfit the training dataset, we would expect the line plot of accuracy on the training set to continue to increase and the test set to rise and then fall again as the model learns statistical noise in the training dataset.

```

1 # plot history
2 pyplot.plot(history.history['accuracy'], label='train')
3 pyplot.plot(history.history['val_accuracy'], label='test')
4 pyplot.legend()
5 pyplot.show()

```

We can tie all of these pieces together, the complete example is listed below.

```

1 # mlp overfit on the two circles dataset
2 from sklearn.datasets import make_circles
3 from keras.layers import Dense
4 from keras.models import Sequential
5 from matplotlib import pyplot
6 # generate 2d classification dataset
7 X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
8 # split into train and test
9 n_train = 30
10 trainX, testX = X[:n_train, :], X[n_train:, :]
11 trainy, testy = y[:n_train], y[n_train:]
12 # define model
13 model = Sequential()
14 model.add(Dense(500, input_dim=2, activation='relu'))
15 model.add(Dense(1, activation='sigmoid'))
16 model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

```

```

17 # fit model
18 history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
19 # evaluate the model
20 _, train_acc = model.evaluate(trainX, trainy, verbose=0)
21 _, test_acc = model.evaluate(testX, testy, verbose=0)
22 print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
23 # plot history
24 pyplot.plot(history.history['accuracy'], label='train')
25 pyplot.plot(history.history['val_accuracy'], label='test')
26 pyplot.legend()
27 pyplot.show()

```

Running the example reports the model performance on the train and test datasets.

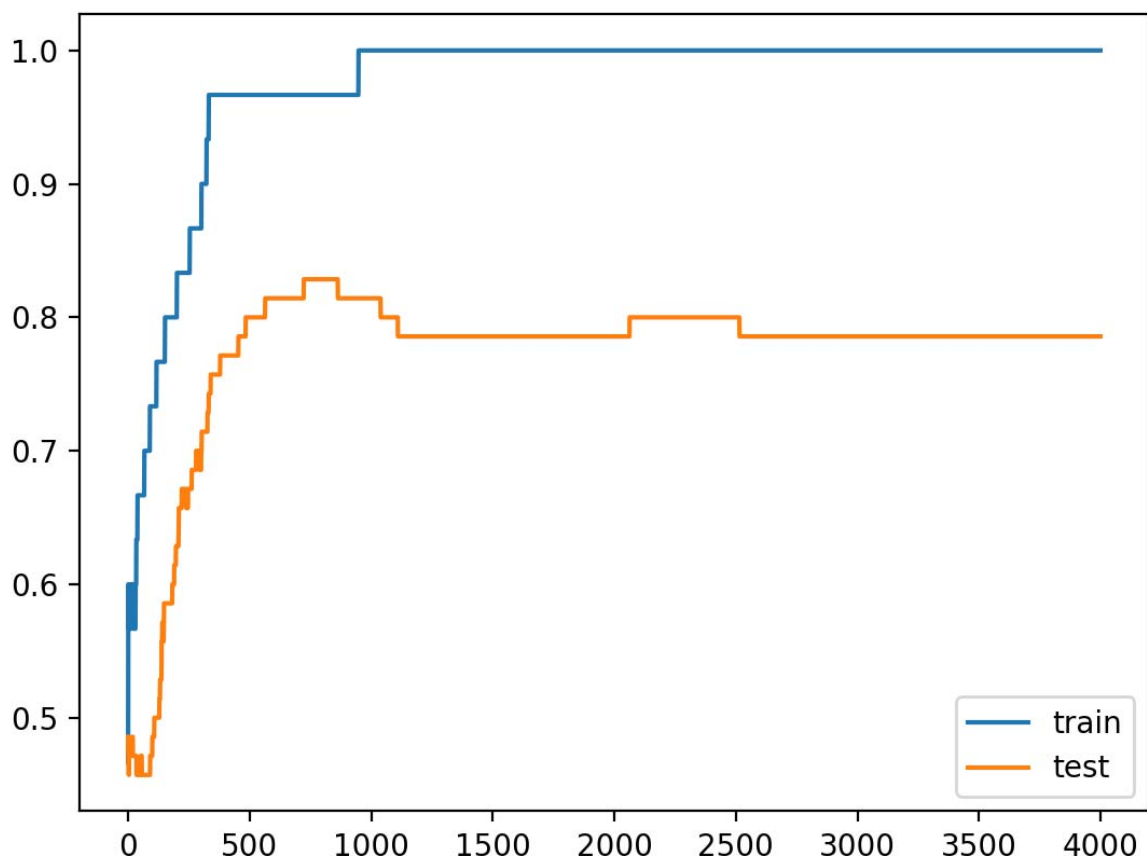
We can see that the model has better performance on the training dataset than the test dataset, one possible sign of overfitting.

Your specific results may vary given the stochastic nature of the neural network and the training algorithm. Because the model is severely overfit, we generally would not expect much, if any, variance in the accuracy across repeated runs of the model on the same dataset.

```
1 Train: 1.000, Test: 0.786
```

A figure is created showing line plots of the model accuracy on the train and test sets.

We can see the expected shape of an overfit model where test accuracy increases to a point and then begins to decrease again.



Line Plots of Accuracy on Train and Test Datasets While Training Showing an Overfit



## Overfit MLP With Activation Regularization

We can update the example to use activation regularization.

There are a few different regularization methods to choose from, but it is probably a good idea to use the most common, which is the [L1 vector norm](#).

This regularization has the effect of encouraging a sparse representation (lots of zeros), which is supported by the rectified linear activation function that permits true zero values.

We can do this by using the `keras.regularizers.l1` class in Keras.

We will configure the layer to use the linear activation function so that we can regularize the raw outputs, then add a relu activation layer after the regularized outputs of the layer. We will set the regularization hyperparameter to 1E-4 or 0.0001, found with a little trial and error.

```
1 model.add(Dense(500, input_dim=2, activation='linear', activity_regularizer=l1(0.0001)))
2 model.add(Activation('relu'))
```

The complete updated example with the L1 norm constraint is listed below:

```
1 # mlp overfit on the two circles dataset with activation regularization
2 from sklearn.datasets import make_circles
3 from keras.layers import Dense
4 from keras.models import Sequential
5 from keras.regularizers import l1
6 from keras.layers import Activation
7 from matplotlib import pyplot
8 # generate 2d classification dataset
9 X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
10 # split into train and test
11 n_train = 30
12 trainX, testX = X[:n_train, :], X[n_train:, :]
13 trainy, testy = y[:n_train], y[n_train:]
14 # define model
15 model = Sequential()
16 model.add(Dense(500, input_dim=2, activation='linear', activity_regularizer=l1(0.0001)))
17 model.add(Activation('relu'))
18 model.add(Dense(1, activation='sigmoid'))
19 model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
20 # fit model
21 history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
22 # evaluate the model
23 _, train_acc = model.evaluate(trainX, trainy, verbose=0)
24 _, test_acc = model.evaluate(testX, testy, verbose=0)
25 print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
26 # plot history
27 pyplot.plot(history.history['accuracy'], label='train')
28 pyplot.plot(history.history['val_accuracy'], label='test')
29 pyplot.legend()
30 pyplot.show()
```

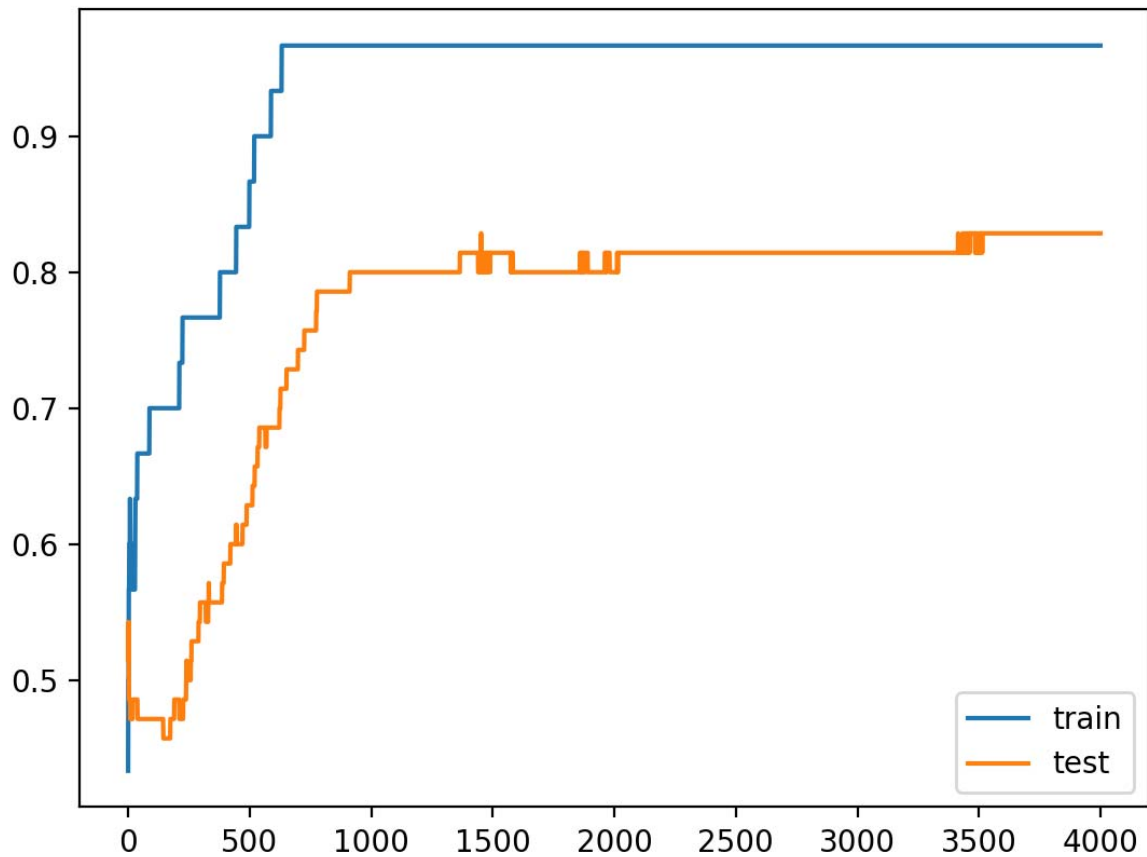
Running the example reports the model performance on the train and test datasets.

We can see that activity regularization resulted in a slight drop in accuracy on the training dataset down from 100% to 96% and a lift in accuracy on the test set up from 78% to 82%.

```
1 Train: 0.967, Test: 0.829
```

Reviewing the line plot of train and test accuracy, we can see that it no longer appears that the model has overfit the training dataset.

Model accuracy on both the train and test sets continues to increase to a plateau.



Line Plots of Accuracy on Train and Test Datasets While Training With Activity Regularization

For completeness, we can compare results to a version of the model where activity regularization is applied after the relu activation function.

```
1 model.add(Dense(500, input_dim=2, activation='relu', activity_regularizer=l1(0.0001)))
```

The complete example is listed below.

```
1 # mlp overfit on the two circles dataset with activation regularization
2 from sklearn.datasets import make_circles
3 from keras.layers import Dense
4 from keras.models import Sequential
5 from keras.regularizers import l1
6 from matplotlib import pyplot
7 # generate 2d classification dataset
8 X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
9 # split into train and test
10 n_train = 30
11 trainX, testX = X[:n_train, :], X[n_train:, :]
12 trainy, testy = y[:n_train], y[n_train:]
13 # define model
14 model = Sequential()
15 model.add(Dense(500, input_dim=2, activation='relu', activity_regularizer=l1(0.0001)))
16 model.add(Dense(1, activation='sigmoid'))
```

```

17 model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
18 # fit model
19 history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
20 # evaluate the model
21 _, train_acc = model.evaluate(trainX, trainy, verbose=0)
22 _, test_acc = model.evaluate(testX, testy, verbose=0)
23 print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
24 # plot history
25 pyplot.plot(history.history['accuracy'], label='train')
26 pyplot.plot(history.history['val_accuracy'], label='test')
27 pyplot.legend()
28 pyplot.show()

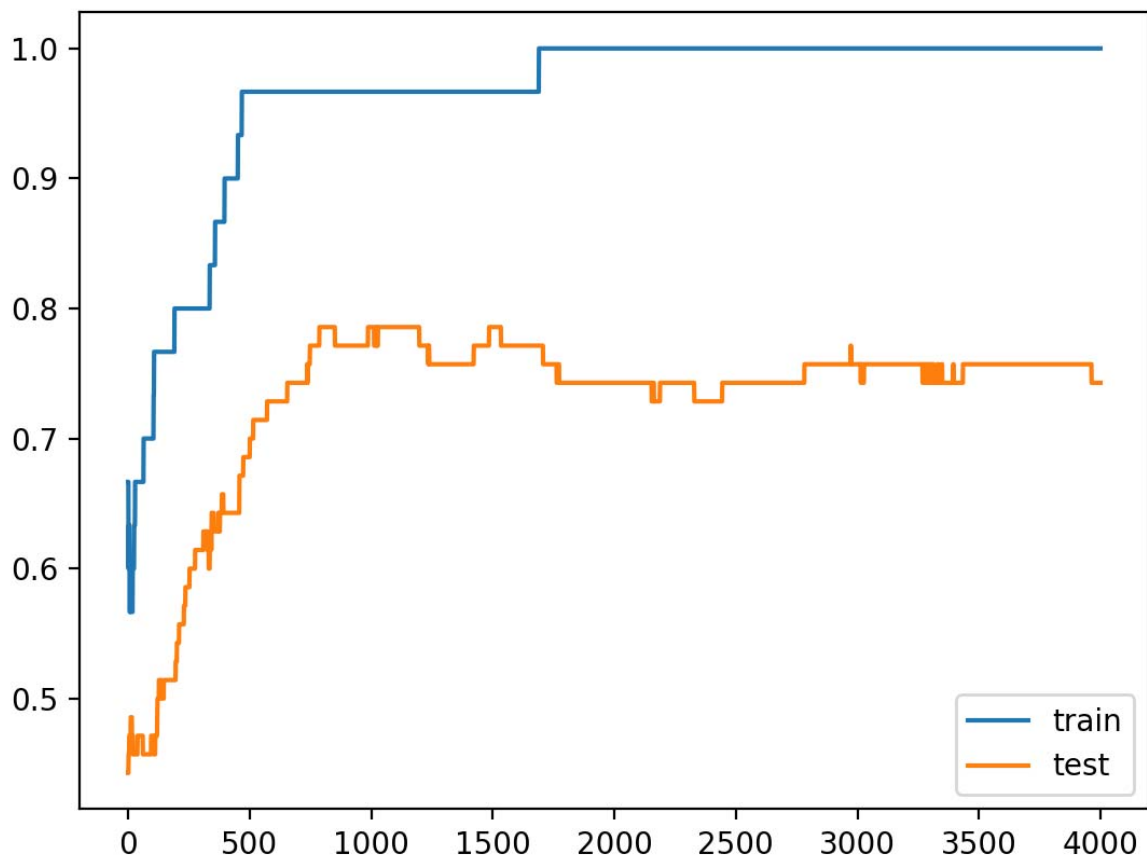
```

Running the example reports the model performance on the train and test datasets.

We can see that, at least on this problem and with this model, activation regularization after the activation function did not improve generalization error; in fact, it made it worse.

```
1 Train: 1.000, Test: 0.743
```

Reviewing the line plot of train and test accuracy, we can see that indeed the model still shows the signs of having overfit the training dataset.



Line Plots of Accuracy on Train and Test Datasets While Training With Activity Regularization, Still Overfit

This suggests that it may be worth experimenting with both approaches for implementing activity regularization with your own dataset, to confirm that you are getting the most out of the method.

## Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Report Activation Mean.** Update the example to calculate the mean activation of the regularized layer and confirm that indeed the activations have been made more sparse.
- **Grid Search.** Update the example to grid search different values for the regularization hyperparameter.
- **Alternate Norm.** Update the example to evaluate the L2 or L1\_L2 vector norm for regularizing the hidden layer outputs.
- **Repeated Evaluation.** Update the example to fit and evaluate the model multiple times and report the mean and standard deviation of model performance.

If you explore any of these extensions, I'd love to know.

## Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### Posts

- [Gentle Introduction to Vector Norms in Machine Learning](#)
- [Building Autoencoders in Keras](#)

### API

- [Keras Regularizers API](#)
- [Keras Core Layers API](#)
- [Keras Convolutional Layers API](#)
- [Keras Recurrent Layers API](#)
- [sklearn.datasets.make\\_circles](#)

## Summary

In this tutorial, you discovered the Keras API for adding activity regularization to deep learning neural network models.

Specifically, you learned:

- How to create vector norm regularizers using the Keras API.
- How to add activity regularization to MLP, CNN, and RNN layers using the Keras API.
- How to reduce overfitting by adding an activity regularization to an existing model.

Do you have any questions?

Ask your questions in the comments below and I will do my best to answer.

---

## Develop Better Deep Learning Models Today!