Use Early Stopping to Halt the Training of Neural Networks At the Right Time

by **Jason Brownlee** on December 10, 2018 in **Deep Learning Performance**

Last Updated on October 3, 2019

A problem with training neural networks is in the choice of the number of training epochs to use.

Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model. Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset.

In this tutorial, you will discover the Keras API for adding early stopping to overfit deep learning neural network models.

After completing this tutorial, you will know:

- How to monitor the performance of a model during training using the Keras API.
- How to create and configure early stopping and model checkpoint callbacks using the Keras API.
- How to reduce overfitting by adding an early stopping to an existing model.

Discover how to train faster, reduce overfitting, and ma n my new book, with 26 step-by-step tutorials and full so

Let's get started.

- **Updated Oct/2019**: Updated for Keras 2.3 and Te

How to Stop Training Deep Neural Networks At the Right Time With Using Early Stopping
Photo by Ian D. Keating, some rights reserved.

Tutorial Overview

This tutorial is divided into six parts; they are:

1. Using Callbacks in Keras
2. Evaluating a Validation Dataset
3. Monitoring Model Performance
4. Early Stopping in Keras
5. Checkpointing in Keras
6. Early Stopping Case Study

Using Callbacks in Keras

Callbacks provide a way to execute code and interact with the training model process automatically.

Callbacks can be provided to the *fit()* function via the "

First, callbacks must be instantiated.

```
...
cb = Callback(...)
```

Then, one or more callbacks that you intend to use mu

```
...
cb_list = [cb, ...]
```

Finally, the list of callbacks is provided to the callback

```
...
model.fit(..., callbacks=cb_list)
```

Evaluating a Validation Dataset in Keras

Early stopping requires that a validation dataset is evaluated during training.

This can be achieved by specifying the validation dataset to the fit() function when training your model.

There are two ways of doing this.

The first involves you manually splitting your training data into a train and validation dataset and specifying the validation dataset to the *fit()* function via the *validation_data* argument. For example:

```
...
model.fit(train_X, train_y, validation_data=(val_x, val_y))
```

Alternately, the *fit()* function can automatically split your training dataset into train and validation sets based on a percentage split specified via the *validation_split* argument.

The *validation_split* is a value between 0 and 1 and defines the percentage amount of the training dataset to use for the validation dataset. For example:

```
...
model.fit(train_X, train_y, validation_split=0.3)
```

In both cases, the model is not trained on the validation dataset. Instead, the model is evaluated on the validation dataset at the end of each training epoch.

Monitoring Model Performance

The loss function chosen to be optimized for your mod

To callbacks, this is made available via the name "*loss*

If a validation dataset is specified to the *fit()* function via the *validation_data* or *validation_split* arguments, then the loss on the validation dataset will be made available via the name "*val_loss*."

Additional metrics can be monitored during the training of the model.

They can be specified when compiling the model via the "*metrics*" argument to the compile function. This argument takes a Python list of known metric functions, such as '*mse*' for mean squared error and '*accuracy*' for accuracy. For example:

```
...
model.compile(..., metrics=['accuracy'])
```

If additional metrics are monitored during training, they are also available to the callbacks via the same name, such as '*accuracy*' for accuracy on the training dataset and '*val_accuracy*' for the accuracy on the validation dataset. Or, '*mse*' for mean squared error on the training dataset and '*val_mse*' on the validation dataset.

Early Stopping in Keras

Keras supports the early stopping of training via a callback called *EarlyStopping*.

This callback allows you to specify the performance measure to monitor, the trigger, and once triggered, it will stop the training process.

The *EarlyStopping* callback is configured when instantiated via arguments.

The "*monitor*" allows you to specify the performance measure to monitor in order to end training. Recall from the previous section that the calculation of measures on the validation dataset will have the '*val_*' prefix, such as '*val_loss*' for the loss on the validation dataset.

```
es = EarlyStopping(monitor='val_loss')
```

Based on the choice of performance measure, the "*mode*" argument will need to be specified as whether the objective of the chosen metric is to increase (maximize or '*max*') or to decrease (minimize or '*min*').

For example, we would seek a minimum for validation loss and a minimum for validation mean squared error, whereas we would seek a maximum for validation accuracy.

```
es = EarlyStopping(monitor='val_loss', mode='m
```

By default, mode is set to '*auto*' and knows that you w

That is all that is needed for the simplest form of early
performance measure stops improving. To discover th
"*verbose*" argument can be set to 1. Once stopped, th

```
es = EarlyStopping(monitor='val_loss', mode='m
```

Often, the first sign of no further improvement may not be the best time to stop training. This is because the model may coast into a plateau of no improvement or even get slightly worse before getting much better.

We can account for this by adding a delay to the trigger in terms of the number of epochs on which we would like to see no improvement. This can be done by setting the "*patience*" argument.

```
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=50)
```

The exact amount of patience will vary between models and problems. Reviewing plots of your performance measure can be very useful to get an idea of how noisy the optimization process for your model on your data may be.

By default, any change in the performance measure, no matter how fractional, will be considered an improvement. You may want to consider an improvement that is a specific increment, such as 1 unit for mean squared error or 1% for accuracy. This can be specified via the "*min_delta*" argument.

```
es = EarlyStopping(monitor='val_accuracy', mode='max', min_delta=1)
```

Finally, it may be desirable to only stop training if performance stays above or below a given threshold or baseline. For example, if you have familiarity with the training of the model (e.g. learning curves) and know that once a validation loss of a given value is achieved that there is no point in continuing training. This can be specified by setting the "*baseline*" argument.

This might be more useful when fine tuning a model, after the initial wild fluctuations in the performance measure seen in the early stages of training a new model are past.

```
es = EarlyStopping(monitor='val_loss', mode='min', baseline=0.4)
```

Checkpointing in Keras

The *EarlyStopping* callback will stop training once triggered, but the model at the end of training may not be the model with best performance on the validation dataset.

An additional callback is required that will save the best model observed during training for later use. This is the *ModelCheckpoint* callback.

The *ModelCheckpoint* callback is flexible in the way it can be used, but in this case we will use it only to save the best model observed during training as defin
validation dataset.

Saving and loading models requires that HDF5 suppo
example, using the *pip* Python installer, this can be ac

```
sudo pip install h5py
```

You can learn more from the h5py Installation docume

The callback will save the model to file, which requires that a path and filename be specified via the first argument.

```
mc = ModelCheckpoint('best_model.h5')
```

The preferred loss function to be monitored can be specified via the monitor argument, in the same way as the *EarlyStopping* callback. For example, loss on the validation dataset (the default).

```
mc = ModelCheckpoint('best_model.h5', monitor='val_loss')
```

Also, as with the *EarlyStopping* callback, we must specify the "*mode*" as either minimizing or maximizing the performance measure. Again, the default is '*auto*,' which is aware of the standard performance measures.

```
mc = ModelCheckpoint('best_model.h5', monitor='val_loss', mode='min')
```

Finally, we are interested in only the very best model observed during training, rather than the best compared to the previous epoch, which might not be the best overall if training is noisy. This can be achieved by setting the "*save_best_only*" argument to *True*.

```
mc = ModelCheckpoint('best_model.h5', monitor='val_loss', mode='min', save_best_only=True)
```

That is all that is needed to ensure the model with the best performance is saved when using early stopping, or in general.

It may be interesting to know the value of the performance measure and at what epoch the model was saved. This can be printed by the callback by setting the "*verbose*" argument to "*1*".

```
mc = ModelCheckpoint('best_model.h5', monitor='val_loss', mode='min', verbose=1)
```

The saved model can then be loaded and evaluated any time by calling the *load_model()* function.

```
# load a saved model
from keras.models import load_model
saved_model = load_model('best_model.h5')
```

Now that we know how to use the early stopping and model checkpoint APIs, let's look at a worked example.

Early Stopping Case Study

In this section, we will demonstrate how to use early s e binary classification problem.

This example provides a template for applying early st on and regression problems.

Binary Classification Problem

We will use a standard binary classification problem th semi-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 "*moons*" dataset because of the shape of the observations in each class when plotted.

We can use the make_moons() 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.

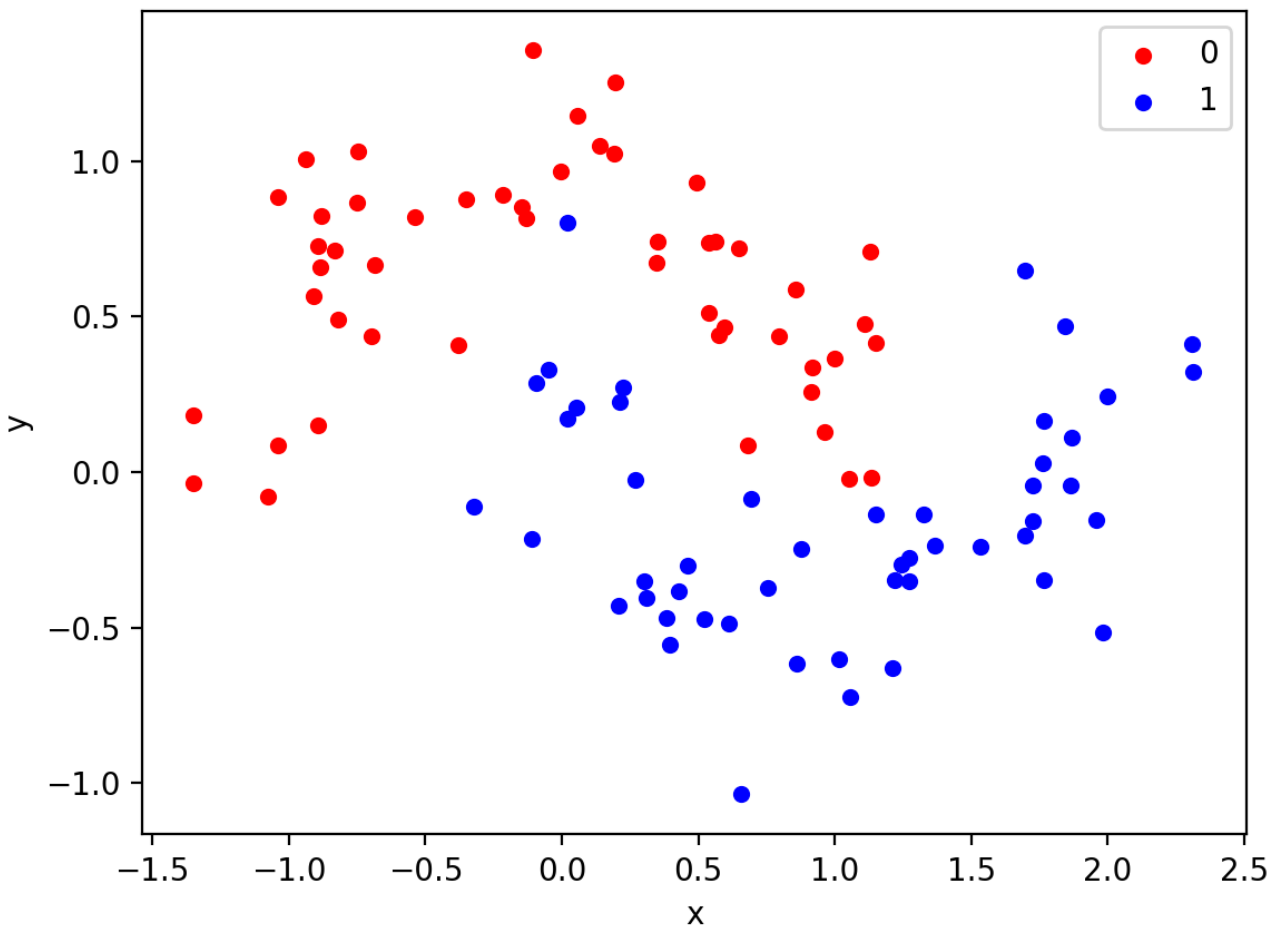
```
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
```

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.

```
# generate two moons dataset
from sklearn.datasets import make_moons
from matplotlib import pyplot
from pandas import DataFrame
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# scatter plot, dots colored by class value
df = DataFrame(dict(x=X[:,0], y=X[:,1], label=y))
colors = {0:'red', 1:'blue'}
fig, ax = pyplot.subplots()
grouped = df.groupby('label')
for key, group in grouped:
    group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])
pyplot.show()
```

Running the example creates a scatter plot showing the semi-circle or moon shape of the observations in each class. We can see the noise in the dispersal of the points making the moons less obvious.



Scatter Plot of Moons 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

The model will have one hidden layer with more node
providing an opportunity to overfit. We will also train th
model overfits.

Before we define the model, we will split the dataset ir
model and 70 to evaluate the fit model's performance.

```
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
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.

```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
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. This is just a simplification for this example. In practice, you would split the training set into train and validation and also hold back a test set for final model evaluation.

```
# fit model
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.

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

Finally, we will plot the loss 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 loss (and 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.

```
# plot training history
pyplot.plot(history.history['loss'], label='tr
pyplot.plot(history.history['val_loss'], label
pyplot.legend()
pyplot.show()
```

We can tie all of these pieces together; the complete e

```
# mlp overfit on the moons dataset
from sklearn.datasets import make_moons
from keras.layers import Dense
from keras.models import Sequential
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, r
```

```
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot training history
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
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.

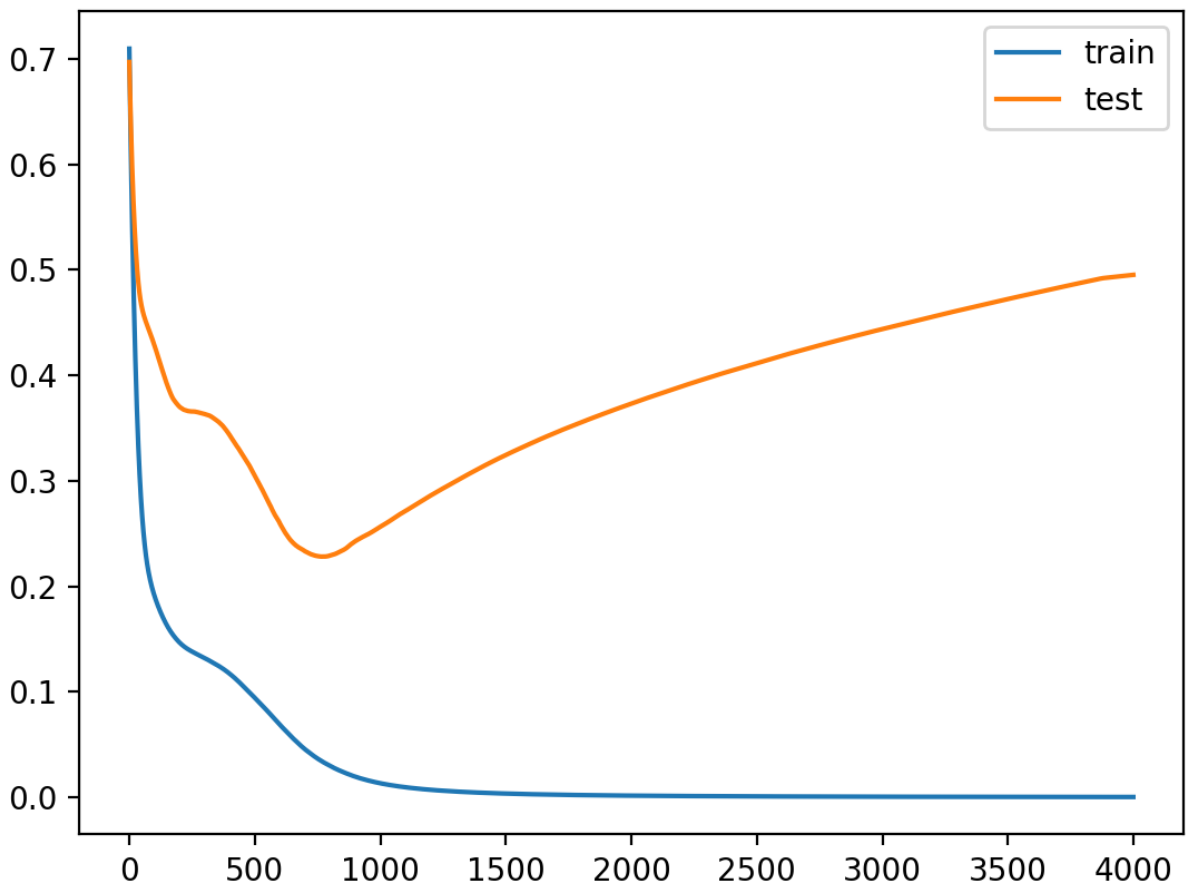
Your specific results may vary given the stochastic nature of the neural network and the training algorithm. Because the model is severely overfit, we generally would not expect much, if any, variance in the accuracy across repeated runs of the model on the same dataset.

```
Train: 1.000, Test: 0.914
```

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

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

Reviewing the figure, we can also see flat spots in the ups and downs in the validation loss. Any early stopping will have to account for these behaviors. We would also expect that a good time to stop training might be around epoch 800.



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

Overfit MLP With Early Stopping

We can update the example and add very simple early stopping.

As soon as the loss of the model begins to increase on the test dataset, we will stop training.

First, we can define the early stopping callback.

```
# simple early stopping
es = EarlyStopping(monitor='val_loss', mode='m
```

We can then update the call to the *fit()* function and sp t.

```
# fit model
history = model.fit(trainX, trainy, validation_ call
```

The complete example with the addition of simple earl

```
# mlp overfit on the moons dataset with simpl
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
```

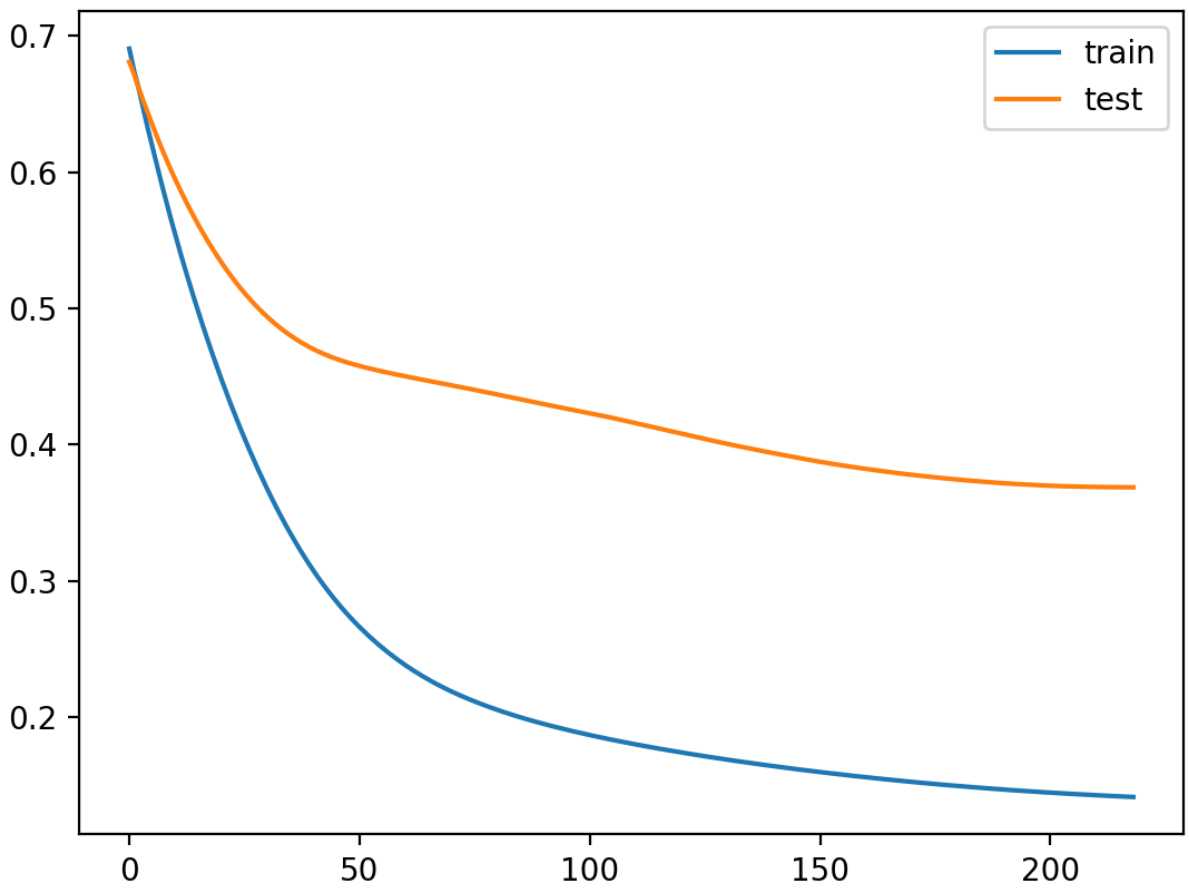
```
from keras.callbacks import EarlyStopping
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# simple early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0, cal
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot training history
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```

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

We can also see that the callback stopped training at epoch 200. This is too early as we would expect an early stop to be around epoch 800. This is also highlighted by the classification accuracy on both the train and test sets, which is worse than no early stopping.

```
Epoch 00219: early stopping
Train: 0.967, Test: 0.814
```

Reviewing the line plot of train and test loss, we can indeed see that training was stopped at the point when validation loss began to plateau for the first time.



Line Plot of Train and Test Loss During Training With Simple Early Stopping

We can improve the trigger for early stopping by waiting a while before stopping.

This can be achieved by setting the "*patience*" argument.

In this case, we will wait 200 epochs before training is stopped. Specifically, this means that we will allow training to continue for up to an additional 200 epochs after the point that validation loss started to degrade, giving the training process an opportunity to get across flat spots or find some additional improvement.

```
# patient early stopping
es = EarlyStopping(monitor='val_loss', mode='m
```

The complete example with this change is listed below

```
# mlp overfit on the moons dataset with patie
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import EarlyStopping
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, r
# split into train and test
```

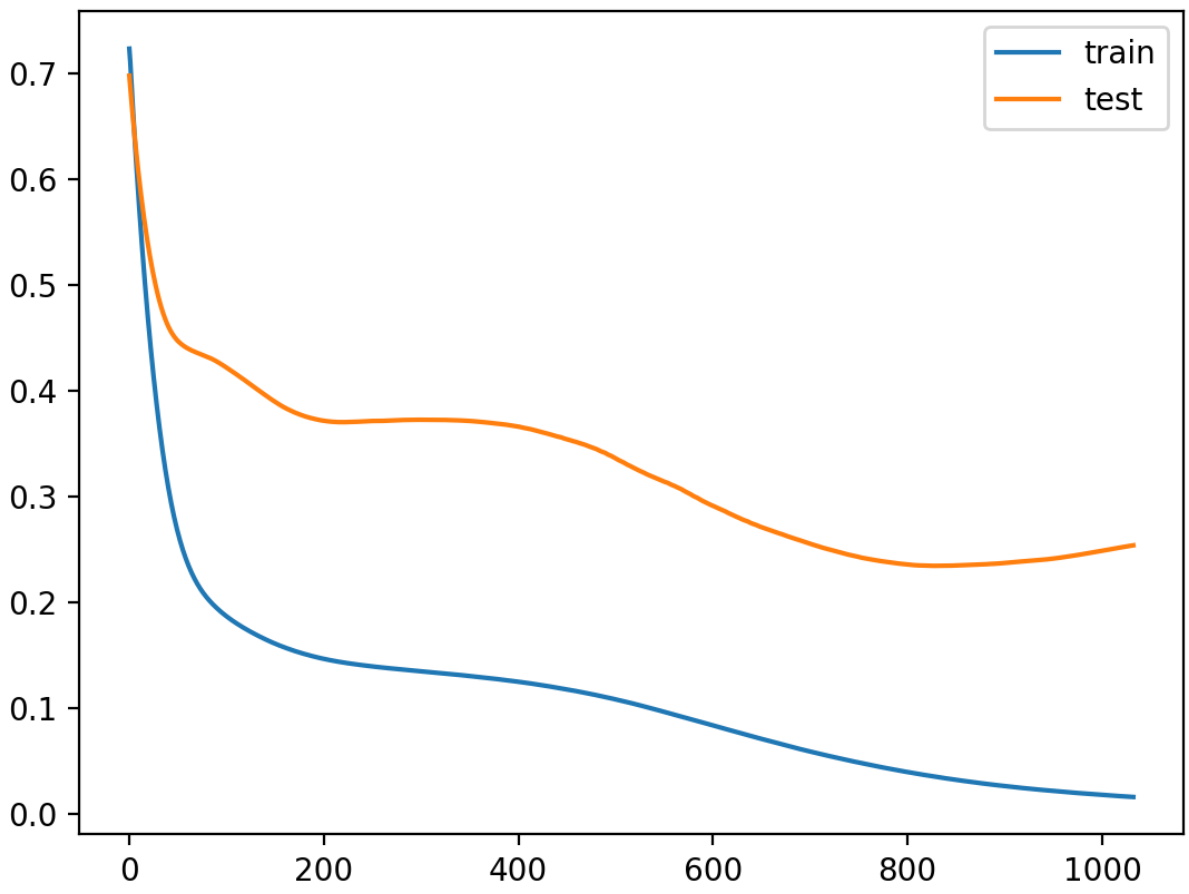
```
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# patient early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=200)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0, cal
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot training history
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```

Running the example, we can see that training was stopped much later, in this case after epoch 1,000. Your specific results may differ given the stochastic nature of training neural networks.

We can also see that the performance on the test dataset is better than not using any early stopping.

```
Epoch 01033: early stopping
Train: 1.000, Test: 0.943
```

Reviewing the line plot of loss during training, we can see that the patience allowed the training to progress past some small flat and bad spots.



Line Plot of Train and Test Loss During Training With Patient Early Stopping

We can also see that test loss started to increase again in the last approximately 100 epochs.

This means that although the performance of the model has improved, we may not have the best performing or most stable model at the end of training. We can address this by using a *ModelChecckpoint* callback.

In this case, we are interested in saving the model with the best accuracy on the test dataset. We could also seek the model with the best loss on the test data … odel with the best accuracy.

This highlights an important concept in model selectio … ay conflict when evaluated using different performance m … ric by which they will be evaluated and presented in the c … n, this will most likely be classification accuracy. Therefo … *ModelCheckpoint* callback to save the best model obs …

```
1 mc = ModelCheckpoint('best_model.h5', monitor= … st_c
```

During training, the entire model will be saved to the file "*best_model.h5*" only when accuracy on the validation dataset improves overall across the entire training process. A verbose output will also inform us as to the epoch and accuracy value each time the model is saved to the same file (e.g. overwritten).

This new additional callback can be added to the list of callbacks when calling the *fit()* function.

```
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0, callba
```

We are no longer interested in the line plot of loss during training; it will be much the same as the previous run.

Instead, we want to load the saved model from file and evaluate its performance on the test dataset.

```
# load the saved model
saved_model = load_model('best_model.h5')
# evaluate the model
_, train_acc = saved_model.evaluate(trainX, trainy, verbose=0)
_, test_acc = saved_model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

The complete example with these changes is listed below.

```
# mlp overfit on the moons dataset with patient early stopping and model checkpointing
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import EarlyStopping
from keras.callbacks import ModelCheckpoint
from matplotlib import pyplot
from keras.models import load_model
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# simple early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=200)
mc = ModelCheckpoint('best_model.h5', monitor='val_accuracy', mode='max', verbose=1, save_best_
# fit model
history = model.fit(trainX, trainy, validatio
# load the saved model
saved_model = load_model('best_model.h5')
# evaluate the model
_, train_acc = saved_model.evaluate(trainX, t
_, test_acc = saved_model.evaluate(testX, tes
print('Train: %.3f, Test: %.3f' % (train_acc,
```

Running the example, we can see the verbose output
new best model is saved and from when no improvem

We can see that the best model was observed at epo
given the stochastic nature of training neural network

Again, we can see that early stopping continued patiently until after epoch 1,000. Note that epoch 880 + a patience of 200 is not epoch 1044. Recall that early stopping is monitoring loss on the validation dataset and that the model checkpoint is saving models based on accuracy. As such, the patience of early stopping started at an epoch other than 880.

```
...
Epoch 00878: val_acc did not improve from 0.92857
Epoch 00879: val_acc improved from 0.92857 to 0.94286, saving model to best_model.h5
Epoch 00880: val_acc did not improve from 0.94286
...
Epoch 01042: val_acc did not improve from 0.94286
Epoch 01043: val_acc did not improve from 0.94286
Epoch 01044: val_acc did not improve from 0.94286
Epoch 01044: early stopping
Train: 1.000, Test: 0.943
```

In this case, we don't see any further improvement in model accuracy on the test dataset. Nevertheless, we have followed a good practice.

Why not monitor validation accuracy for early stopping?

This is a good question. The main reason is that accuracy is a coarse measure of model performance during training and that loss provides more nuance when using early stopping with classification problems. The same measure may be used for early stopping and model checkpointing in the case of regression, such as mean squared error.

Extensions

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

- **Use Accuracy**. Update the example to monitor accuracy on the test dataset rather than loss, and plot learning curves showing accuracy.
- **Use True Validation Set**. Update the example to split the training set into train and validation sets, then evaluate the model on the test dataset.
- **Regression Example**. Create a new example of using early stopping to address overfitting on a simple regression problem and monitoring mean squared error.

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

Further Reading

This section provides more resources on the topic if y

Posts

- Avoid Overfitting by Early Stopping With XGBoos
- How to Check-Point Deep Learning Models in Ke

API

- H5Py Installation Documentation
- Keras Regularizers API
- Keras Core Layers API
- Keras Convolutional Layers API
- Keras Recurrent Layers API
- Keras Callbacks API
- sklearn.datasets.make_moons API

Summary

In this tutorial, you discovered the Keras API for adding early stopping to overfit deep learning neural network models.

Specifically, you learned:

- How to monitor the performance of a model during training using the Keras API.
- How to create and configure early stopping and model checkpoint callbacks using the Keras API.
- How to reduce overfitting by adding a early stopping to an existing model.

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

About Jason Brownlee

Jason Brownlee, PhD is a machine learning specialist who teaches developers how to get results with modern machine learning methods via hands-on tutorials.

View all posts by Jason Brownlee →

‹ A Gentle Introduction to Early Stopping to Avoid Overtraining Neural Networks

Train Neural Networks With Noise to Reduce Overfitting ›

62 Responses to *Use Early Stopping to Halt the Training of Neural Networks At the Right Time*

Walid Ahmed December 12, 2018 at 7:07 am #

REPLY

Thanks a lot, A very clear illustration as always

Jason Brownlee December 12, 2018 at 2:10 pm #

REPLY

Thanks, I'm glad it helped.

Theekshana December 21, 2018 at 5:54 pm #

REPLY

Hi Jason,

Is this process of early stopping should be done after we selected the suitable architecture for the neural network (Maybe using cross-validation scores of different architectures.)?

Thank you for the tutorial. Big fan of this site 🙂

Jason Brownlee December 22, 2018 at 6:02

Perhaps, you can start with an over-speci

stopping immediately.

Khalil January 16, 2019 at 12:46 am #

Wow, this is a lovely focused and lucid explanation of callbacks. Thanks for writing this Jason!

Jason Brownlee January 16, 2019 at 5:48 am #

REPLY

Thanks, I'm glad it helped!

Daniel Penalva February 20, 2019 at 2:39 am #

REPLY

Hi Jason, this is stupendous article. Congratulations.

Still reading and has a lot to get from him.

In november i posted in another post of yours about checkpoints, my main objective at that time (and still is) is to do hyperparam optimization with checkpoints.

Do you think the Early Stopping approach can achieve this by Stopping->Reloading->Compiling->Continue Training->Hyperparam change->Stopping->So on …

??

Thank you in advance.

Jason Brownlee February 20, 2019 at 8:10 am #

REPLY

I generally don't recommend using early stopping with hyperparameter tuning, it makes the process messy.

Tune parameters than add early stopping as a regularization method at the end – my best advice.

Daniel Penalva February 21, 2019 at 2:57 am #

REPLY

But, to use Early Stopping, isnt the sa ?

My point is: I wanna to be able to reload the sa
change model hyperparams (Archs, Batch Siz
as before (or same dataset, as if i were forced

The Whole point maybe is: Is there in Keras a
(that doesnt work too well with Keras checkpoi

Thank you …again.

Jason Brownlee February 21, 2019 at 8:15 am #

REPLY

Yes, I teach writing the for-loop for parameter testing manually:
https://machinelearningmastery.com/how-to-grid-search-deep-learning-models-for-time-series-forecasting/

Deniz May 6, 2019 at 3:03 pm #

REPLY

how about Dropout? Do the rules for weight decay also apply to Dropout?

Jason Brownlee May 7, 2019 at 6:09 am #

REPLY

In what way exactly?

Fazano March 24, 2019 at 2:58 pm #

REPLY

Hi Mr. Jason

I use early stopping for LSTM forecasting. when I ran training for the first time, early stopping stopped at Epoch 20, and when I run it it stopped for the second time in Epoch 17. It was always different every time. can you explain why that happened?

my code is like this:

```
model = Sequential()
model.add(LSTM(4,activation='tanh',input_shape=(1,3),recurrent_activation= 'hard_sigmoid'))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='adam',metrics = [metrics.mae])
early_stop = EarlyStopping(monitor='loss', patience=2, verbose=0, mode='min')
model.fit(X_train,Y_train,epochs= 100 ,batch_size= 1,verbose=2, callbacks=[early_stop])
```

thank you

Jason Brownlee March 25, 2019 at 6:41 am #

Yes, this is due to the stochastic learning

You can learn more here:
https://machinelearningmastery.com/faq/single-faq
code

KK March 27, 2019 at 2:43 pm #

REPLY

Hi Dr. Jason,

Thanks for the excellent post and helping in understanding DL related topics. I have a question.

When we use ModelCheckpoint whether only the weights would be saved or the complete model (along with the model architecture)

Whenever I try to save the model and load it directly in a different program it throws up and error (Error details will post soon).
saved_model = load_model('best_model.h5')

So what I had to follow the below approach:
1. First build the model architecture .
2. Then load the saved weight (or model, I am using the .h5 file saved using ModelCheckpoint)

Is this the right approach? I can share the github link so that you can have a look at my code if everything is fine.

Thank you,
KK

Jason Brownlee March 28, 2019 at 8:07 am #

REPLY

I recommend using a function to create your model architecture again, then load the weights directly.

Chunhui March 29, 2019 at 3:24 am #

REPLY

Hi Jason, thank you very much for your post, it is very useful. But in my model, I can not use the ModelCheckPoints, the error says the loss is unknown, so I guess the Model checkpoints is not suitable for the model with customized loss function?

Jason Brownlee March 29, 2019 at

Perhaps your model is mis-config

KK April 1, 2019 at 4:25 pm #

Hi Jason, Thanks for the clarification.
a separate code to create the model architectu

JStocks March 30, 2019 at 3:15 am #

REPLY

Thank you Jason!

I have a question. Let's say I have gone through the full round of hyperparameter tuning with kfold cross validation.. and settle on an architecture, parameters, etc. I am ready to use the final model externally. Would I just take one of the trained models from one of the folds? Or.. Would you train the final model on all of the data? With a validation set, you have an indication of when it starts to overfit, while training with all of the data means the models gets to see more data.

The problem is if you train on all of the data.. it's hard to know at what epoch to stop training.

What would you suggest?

Jason Brownlee March 30, 2019 at 6:32 am #

REPLY

No, I generally recommend fitting a new final model on all available data with the best config.

If you want to use early stopping, some data will need to be held back for the validation set.

Esra Karasu May 13, 2019 at 10:10 pm #

REPLY

I tried to load the saved model with "load_model()" code but I got this error:

load_model() missing 1 required positional argument: 'filepath'

can you please help me with this errror?

Jason Brownlee May 14, 2019 at 7:43 am #

REPLY

You must enter the path to the model file to load

Sourav June 11, 2019 at 3:31 pm #

Hi Jason,

I was wondering if there there is any hard and bound r
stopping. Say, I use 'validation accuracy' i.e. if the vali
stop training my model. What are the pros and cons of

Jason Brownlee June 12, 2019 at 7:50 am #

REPLY

Typically a rise in loss on a representative validation dataset is used as the stopping condition.

hayj June 14, 2019 at 8:01 pm #

REPLY

Hello Jason, I don't understand if early stopping with patience of 4 (for example) will stop training when val_loss_i > val_loss_i -1 > val_loss_i -2 > val_loss_i -3 (a sequence of loss increasing)
Or will stop training when val_loss_i > min_overall_val_loss & val_loss_i-1 > min_overall_val_loss & val_loss_i-2 > min_overall_val_loss & val_loss_i-3 > min_overall_val_loss (a sequence that independently do not perform better than the overall min loss)

Jason Brownlee June 15, 2019 at 6:32 am #

REPLY

I don't follow, sorry. What problem are you having exactly?

hayj June 18, 2019 at 5:40 am #

REPLY

Sorry I can explain better: do the early stopping mecanism takes into account the minimum overall loss (from all previous epochs) ? Or just takes into account the current loss until the loss of the current epoch – patience ?

Jason Brownlee June 18, 2019 at 6:43 am #

REPLY

You configure the early stopping however you need.

SSN June 20, 2019 at 2:36 pm #

Hello Jason,

Thank you for all your amazing notes. I have a questio
training, testing and validation data sets. I also want to
for each epoch. Is it possible in Keras?

SSN June 20, 2019 at 4:11 pm #

Or in simpler words can I do like this:
1. Split data into training and testing
2. Split the training data to training and validation. (So finally I have three sets, testing kept aside for the end)
3. Now fit a model for training data, use validation data and predict and get the model accuracy
4. If model accuracy is less than some required number go back to step to step 3 and re shuffle and get a new combination of another random training and validation datasets. Use the previous model and weights, improvise this or increment the weights from this state
5. Do this till a decent accuracy with validation is achieved
6. Then use the test data to get the final accuracy numbers

My main questions are , is this effective way of doing it? and are the weights saved when I use the fit() again, by which I mean are the weights tuned from the previous existing value everytime I use fit() command?

REPLY

Jason Brownlee June 21, 2019 at 6:33 am #

Yes, the weights remain across calls to fit()

REPLY

Jason Brownlee June 21, 2019 at 6:32 am #

Yes, but you will have to run the training process manually, e.g. each loop and call train_on_batch() on your model or fit() with 1 epoch.

I have a few examples on the blog I believe.

REPLY

SSN June 21, 2019 at 2:04 pm #

Thank you again Jason.
I did search for those on your blog. Didn't get a direct answer for those questions. I guess your answers helped me to get one. Will implement this and see how it turns out. Thanks a lot for the tons of information in your blogs.

Jason Brownlee June 22, 2019 at

Let me know how you go.

SSN June 21, 2019 at 7:27 pm #

Hi Jason,

Please check the script below if I have missed anything.

Standardize the input and output data and

1. Initialize model (compile)
2. fit model
3. save model

In loop

4. Load the saved model
5. shuffle data between training and validation sets
6. model_int.fit(X_train, Y_train, epochs = 1,batch_size = 10, verbose = 1)
7. model_int.save('intermediate_model.h5')
8. Predict Y using validation X data
9. Compare predicted Y data and actual Y data
10. Decide whether to back to step 4 or end based on step 7's output.

Did I miss anything? Also saving in step 6, does it save the last batch model or the model a result of all the batches? Or should I run with batch size 1 and save after every batch and re iterate from there ?

Jason Brownlee June 22, 2019 at 6:36 am #

REPLY

Not sure I follow, why would you fit then fit again?

SSN June 24, 2019 at 2:09 pm #

REPLY

I fit it with different sets of training and validation data sets. I keep aside a part of the data for final testing which I call test set. Then in the remaining instead of using the same training set I use different combinations of training and testing sets until the prediction shows good metrics. Once it is, I use the validation set to see the final metrics.

Jason Brownlee June 24, 2019 at 2:30 pm #

Interesting approach.

Tejas June 28, 2019 at 1:14 pm #

Hi,

I was trying to stop the model early based on the base

below command to monitor the validation loss is not working. I also tried with patience even that is not working. The model's initial validation loss is around 21.

"

```
model.compile(loss='categorical_crossentropy', optimizer=Adam(lr=0.000016), metrics=['accuracy'])
es = EarlyStopping(monitor='val_loss', mode='min', baseline=0.5)
class_weight = {0: 0.31,
1: 0.69}
history=model.fit(x_train, y_train, validation_data=(x_validation,y_validation),
epochs=150, batch_size=32,class_weight=class_weight, callbacks=[es])
```

"

The model is stopping after one epoch. I searched everywhere, but couldn't figure out the problem. I appreciate any help. Thanks

Jason Brownlee June 28, 2019 at 1:56 pm #

REPLY

I'm eager to help, but I don't have the capacity to debug your example sorry.

Christian June 30, 2019 at 7:43 pm #

REPLY

Hi Tejas,

that might be because the baseline parameter is explained incorrectly in the article. Rather then stopping if the model reaches the value, it's the opposite: the model stops if it does NOT reach that value, as can be seen in the keras documentation. I think the patience parameter controlls how many epochs the model has to reach the baseline before stopping.

Gunnar June 28, 2019 at 11:07 pm #

REPLY

Dear Jason.

Many thanks for all the effort you've put into this page, helpful.

I have some trouble deciding how many epochs I shou optimal configuration of my model I used early stoppin a final model I want to train on all the available data, s generating the final models.

Do you have any suggestions as to how one should de training a final model? Is it reasonable to use number the training when I was configuring the model?

Jason Brownlee June 29, 2019 at 6:55 am #

REPLY

You can use early stopping, run a few times, note the number of epochs each time it is stopped and use the average when fitting the model on all data.

Shark August 3, 2019 at 7:00 pm #

REPLY

Thank you! This article is really helpful!

Jason Brownlee August 4, 2019 at 6:25 am #

REPLY

Thanks, I'm glad it helped.

michelle October 19, 2019 at 3:27 am #

REPLY

Hi Jason,

Thank you so much for the great explanation!

In the case of binary classification, I sometimes run into the scenario where validation loss starts to increase while the validation accuracy is still improving (Test accuracy also improves). I think this is because the model is still improving in predicting the labels, even though the actual loss value is getting bigger.

Can I use the model that has bigger validation accuracy (also better test accuracy) but bigger validation loss? Since our final goal is to have better prediction of labels, why do we care about increasing in loss?

Jason Brownlee October 19, 2019 at 6:49 am #

REPLY

Sure.

But perhaps test to ensure the finding is robust. A

David October 20, 2019 at 2:44 am #

Thanks for the article. When using val_acc in
model out with the model corresponding to the highest
model to predict against my validation set as a check,
uses transfer learning on NasNet.

Do you know why that might be?

Jason Brownlee October 20, 2019 at 6:21 am #

REPLY

The accuracy calculated on val during early stopping might be an average over the batches – it is an estimate.

Perhaps try running early stopping a few times, and ensemble the collection of final models to reduce the variance in their performance?

Aditya December 6, 2019 at 5:35 pm #

REPLY

model = Sequential()
model.add(LSTM(neurons, batch_input_shape=(batch_size, X_train.shape[1], X_train.shape[2]), activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')

global_trial = global_trial[0] + 1
model_path = "../deeplearning/saved_models_2/" + str(name) + "/" + str(global_trial) + "_" + "model.h5"

# simple early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=50)
mc = ModelCheckpoint(model_path, monitor='val_loss', mode='min', verbose=1, save_best_only=True)

# Fit the model
history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=nb_epochs, batch_size=1, verbose=0,
callbacks=[es, mc])

# load the saved model
saved_model = load_model(model_path)

evaluate the model
_, train_loss = saved_model.evaluate(X_train, y_train, verbose=0)
_, test_loss = saved_model.evaluate(X_test, y_test, ve
print('Train: %.3f, Test: %.3f' % (train_loss, test_loss))

Error = tensorflow.python.framework.errors_impl.Invali
a tensor with shape [32,1]

while I am using same train and test data for Evaluatio

Jason Brownlee December 7, 2019 at 5:35 a

The error suggest the shape of the data does not match the shape expected by the model.

Mark Ryan December 9, 2019 at 6:15 am #

REPLY

Fantastic article – extremely clear with succinct examples that are easy to follow. Thanks very much for sharing this.

Jason Brownlee December 9, 2019 at 6:55 am #

REPLY

Thanks!

Murilo Souza January 27, 2020 at 9:34 pm #

REPLY

Great article again! Thanks for the information Jason!

I learn here in your site more than i learn with my professors in classroom lol. If you're not already, i bet you would be a great professor!

Jason Brownlee January 28, 2020 at 7:52 am #

REPLY

Thanks!

farukgogh February 5, 2020 at 4:24 am #

REPLY

hi Jason, how can I visualize graph when using model.fit_generator? how will I use history to visualize graph? such as

model.fit_generator(
train_generator,
steps_per_epoch=… ,
epochs=….,
validation_data=validation_generator,
validation_steps= …)

Jason Brownlee February 5, 2020 at 8:20 am

You might have to save or log the history

farukgogh February 6, 2020 at 2:51 am #

REPLY

can you explain in more detail, please?

Jason Brownlee February 6, 2020 at 8:32 am #

REPLY

Sure, you could setup a callback that then appends loss scores to a file.

It is an engineering question with many solutions. Perhaps prototype a few and see what works.

James McGuigan March 7, 2020 at 6:58 am #

REPLY

Its worth noting that as an alterative to ModelCheckpointing, there is a restore_best_weights parameter:

tf.keras.callbacks.EarlyStopping(restore_best_weights=True)

restore_best_weights: Whether to restore model weights from the epoch with the best value of the monitored quantity. If False, the model weights obtained at the last step of training are used.

– https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

Jason Brownlee March 7, 2020 at 7:23 am #

REPLY

Thanks, looks new and tf.keras.

Leave a Reply

Name (required)

Email (will not be published) (required)

Website

SUBMIT COMMENT

Welcome!

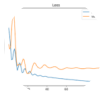
My name is *Jason Brownlee* PhD, and I **help developers** get results with **machine learning**.

Read more

Never miss a tutorial:

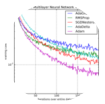
Picked for you:

How To Improve Deep Learning Performance

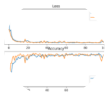


How to use Learning Curves to Diagnose Machine Learning Model Performance

How to Develop a Stacking Ensemble for Deep Learning Neural Networks in Python With Keras



Gentle Introduction to the Adam Optimization Algorithm for Deep Learning



How to Choose Loss Functions When Training Dee

Loving the

The Better Deep L

where I keep the *B*

Start Machine Learning

You can master applied Machine Learning **without math or fancy degrees**.
Find out how in this *free* and *practical* course.

Email Address

START MY EMAIL COURSE

SEE WHAT'S INSIDE