How to Develop an Ensemble of Deep Learning Models in Keras

by **Jason Brownlee** on December 21, 2018 in **Deep Learning Performance**

Last Updated on January 10, 2020

Deep learning neural network models are highly flexible nonlinear algorithms capable of learning a near infinite number of mapping functions.

A frustration with this flexibility is the high variance in a final model. The same neural network model trained on the same dataset may find one of many different possible "good enough" solutions each time it is run.

Model averaging is an ensemble learning technique that reduces the variance in a final neural network model, sacrificing spread in the performance of the model for a confidence in what performance to expect from the model.

In this tutorial, you will discover how to develop a model averaging ensemble in Keras to reduce the variance in a final model.

After completing this tutorial, you will know:

- Model averaging is an ensemble learning techniq nce of deep learning neural network models.
- How to implement model averaging in Keras for c problems.
- How to work through a multi-class classification p variance of the final model.

Discover how to train faster, reduce overfitting, and m in my new book, with 26 step-by-step tutorials and full s

Let's get started.

- **Updated Oct/2019**: Updated for Keras 2.3 and TensorFlow 2.0.
- **Updated Jan/2020**: Updated for changes in scikit-learn v0.22 API.

How to Reduce the Variance of Deep Learning Models in Keras With Model Averaging Ensembles
Photo by John Mason, some rights reserved.

Tutorial Overview

This tutorial is divided into six parts; they are:

1. Model Averaging
2. How to Average Models in Keras
3. Multi-Class Classification Problem
4. MLP Model for Multi-Class Classification
5. High Variance of MLP Model
6. Model Averaging Ensemble

Model Averaging

Deep learning neural network models are nonlinear m m.

This means that they are highly flexible, capable of lea nd approximating any mapping function, given enough re

models suffer high variance.

This means that the models are highly dependent on the specific training data used to train the model and on the initial conditions (random initial weights) and serendipity during the training process. The result is a final model that makes different predictions each time the same model configuration is trained on the same dataset.

This can be frustrating when training a final model for use in making predictions on new data, such as operationally or in a machine learning competition.

The high variance of the approach can be addressed by training multiple models for the problem and combining their predictions. This approach is called model averaging and belongs to a family of techniques called ensemble learning.

How to Average Models in Keras

The simplest way to develop a model averaging ensemble in Keras is to train multiple models on the same dataset then combine the predictions from each of the trained models.

Train Multiple Models

Training multiple models may be resource intensive, depending on the size of the model and the size of the training data.

You may have to train the models sequentially on the
worth training the models in parallel using cloud infras

The number of models required for the ensemble may
model. A benefit of the approach is that you can conti
and evaluate their impact on the performance by mak

For small models, you can train the models sequentia
experiment. For example:

```
...
# train models and keep them in memory
n_members = 10
models = list()
for _ in range(n_members):
    # define and fit model
    model = ...
    # store model in memory as ensemble member
    models.add(models)
...
```

For large models, perhaps trained on different hardware, you can save each model to file.

```
...
# train models and keep them to file
n_members = 10
for i in range(n_members):
    # define and fit model
    model = ...
    # save model to file
    filename = 'model_' + str(i + 1) + '.h5'
    model.save(filename)
    print('Saved: %s' % filename)
...
```

Models can then be loaded later.

Small models can all be loaded into memory at the same time, whereas very large models may have to be loaded one at a time to make a prediction, then later to have the predictions combined.

```
from keras.models import load_model
...
# load pre-trained ensemble members
n_members = 10
models = list()
for i in range(n_members):
    # load model
    filename = 'model_' + str(i + 1) + '.h5'
    model = load_model(filename)
    # store in memory
    models.append(model)
...
```

Combine Predictions

Once the models have been prepared, each model can be used to make a prediction and the predictions can be combined.

In the case of a regression problem where each mode be collected and the average calculated.

```
...
# make predictions
yhats = [model.predict(testX) for model in mode
yhats = array(yhats)
# calculate average
outcomes = mean(yhats)
```

In the case of a classification problem, there are two c

The first is to calculate the mode of the predicted integer class values.

```
...
# make predictions
yhats = [model.predict_classes(testX) for model in models]
yhats = array(yhats)
# calculate mode
outcomes, _ = mode(yhats)
```

A downside of this approach is that for small ensembles or problems with a large number of classes, the sample of predictions may not be large enough for the mode to be meaningful.

In the case of a binary classification problem, a sigmoid activation function is used on the output layer and the average of the predicted probabilities can be calculated much like a regression problem.

In the case of a multi-class classification problem with more than two classes, a softmax activation function is used on the output layer and the sum of the probabilities for each predicted class can be calculated before taking the argmax to get the class value.

```
...
# make predictions
yhats = [model.predict(testX) for model in models]
yhats = array(yhats)
# sum across ensembles
summed = numpy.sum(yhats, axis=0)
# argmax across classes
outcomes = argmax(summed, axis=1)
```

These approaches for combining predictions of Keras models will work just as well for Multilayer Perceptron, Convolutional, and Recurrent Neural Networks.

Now that we know how to average predictions from multiple neural network models in Keras, let's work through a case study.

Multi-Class Classification Problem

We will use a small multi-class classification problem as the basis to demonstrate a model averaging ensemble.

The scikit-learn class provides the make_blobs() function that can be used to create a multi-class classification problem with the prescribed number of s
samples within a class.

We use this problem with 500 examples, with input va
points) and a standard deviation of 2.0 for points with
(seed for the pseudorandom number generator) to ens

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_f
```

The results are the input and output elements of a dat

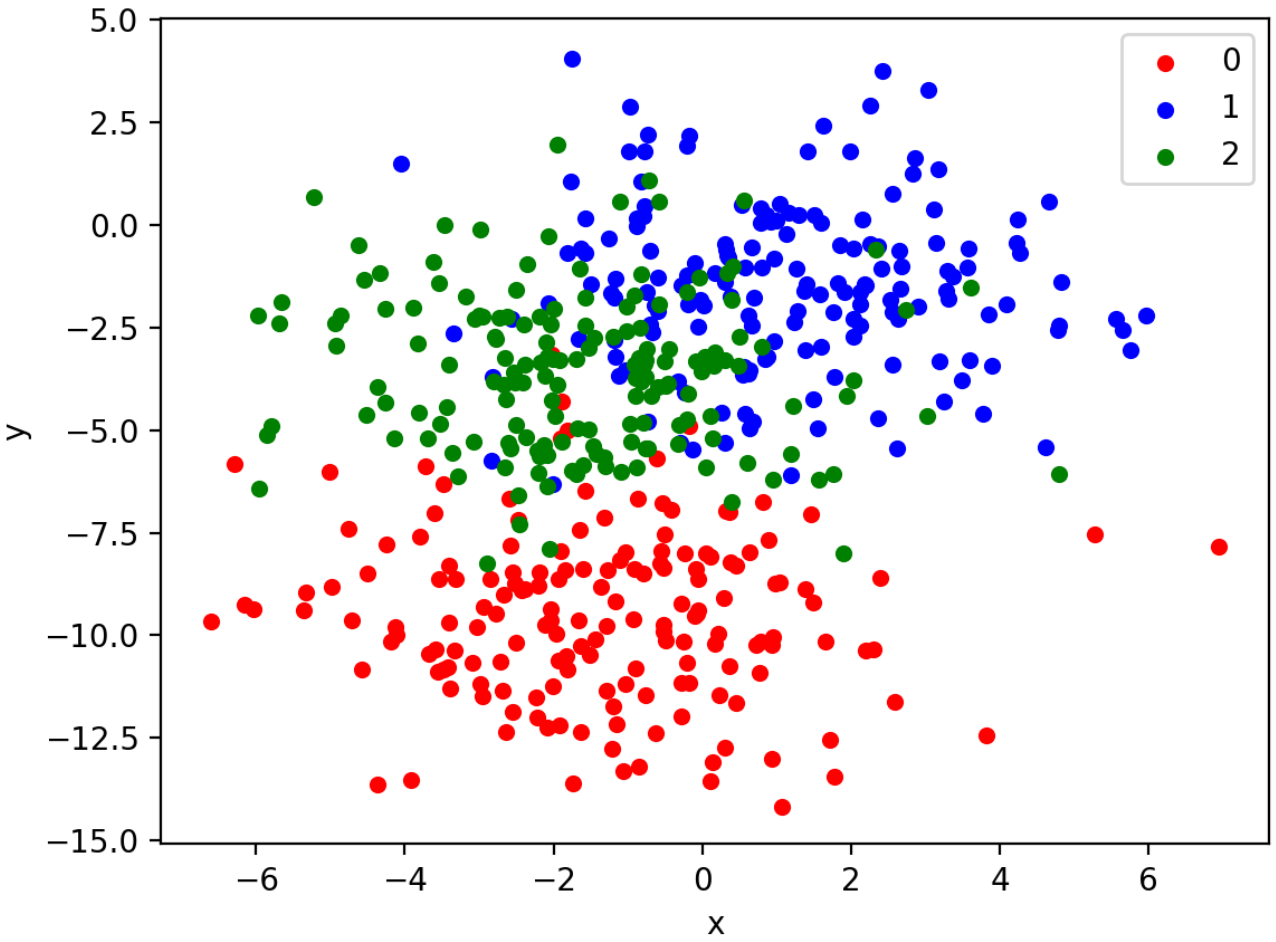
In order to get a feeling for the complexity of the problem, we can graph each point on a two-dimensional scatter plot and color each point by class value.

The complete example is listed below.

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
from pandas import DataFrame
# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_features=2, cluster_std=2, random_state=2)
# scatter plot, dots colored by class value
df = DataFrame(dict(x=X[:,0], y=X[:,1], label=y))
colors = {0:'red', 1:'blue', 2:'green'}
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 of the entire dataset. We can see that the standard deviation of 2.0 means that the classes are not linearly separable (separable by a line) causing many ambiguous points.

This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different "*good enough*" candidate solutions resulting in a high variance.



Scatter Plot of Blobs Dataset with Three Classes and Points Colored by Class Value

MLP Model for Multi-Class Classification

Now that we have defined a problem, we can define a model to address it.

We will define a model that is perhaps under-constrained and not tuned to the problem. This is intentional to demonstrate the high variance of a neural network model seen on truly large and challenging supervised learning problems.

The problem is a multi-class classification problem, ar
function on the output layer. This means that the mod
probability that the sample belongs to each of the 3 c
the class values.

```
1 y = to_categorical(y)
```

Next, we must split the dataset into training and test s
performance of the model and to plot its performance
30% of the data for training and 70% for the test set.

This is an example of a challenging problem where we have more unlabeled examples than we do labeled examples.

```
# split into train and test
n_train = int(0.3 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Next, we can define and compile the model.

The model will expect samples with two input variables. The model then has a single hidden layer with 15 modes and a rectified linear activation function, then an output layer with 3 nodes to predict the probability of each of the 3 classes and a softmax activation function.

Because the problem is multi-class, we will use the categorical cross entropy loss function to optimize the model and the efficient Adam flavor of stochastic gradient descent.

```
# define model
model = Sequential()
model.add(Dense(15, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

The model is fit for 200 training epochs and we will evaluate the model each epoch on the test set, using the test set as a validation set.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
```

At the end of the run, we will evaluate the performance of the model on both the train and the test sets.

```
# 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))
```

Then finally, we will plot learning curves of the model accuracy over each training epoch on both the training and test dataset.

```
# plot history
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

The complete example is listed below.

```
# fit high variance mlp on blobs classificati
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_
y = to_categorical(y)
# split into train and test
```

```
n_train = int(0.3 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(15, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, 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))
# learning curves of model accuracy
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Running the example first prints the performance of the final model on the train and test datasets.

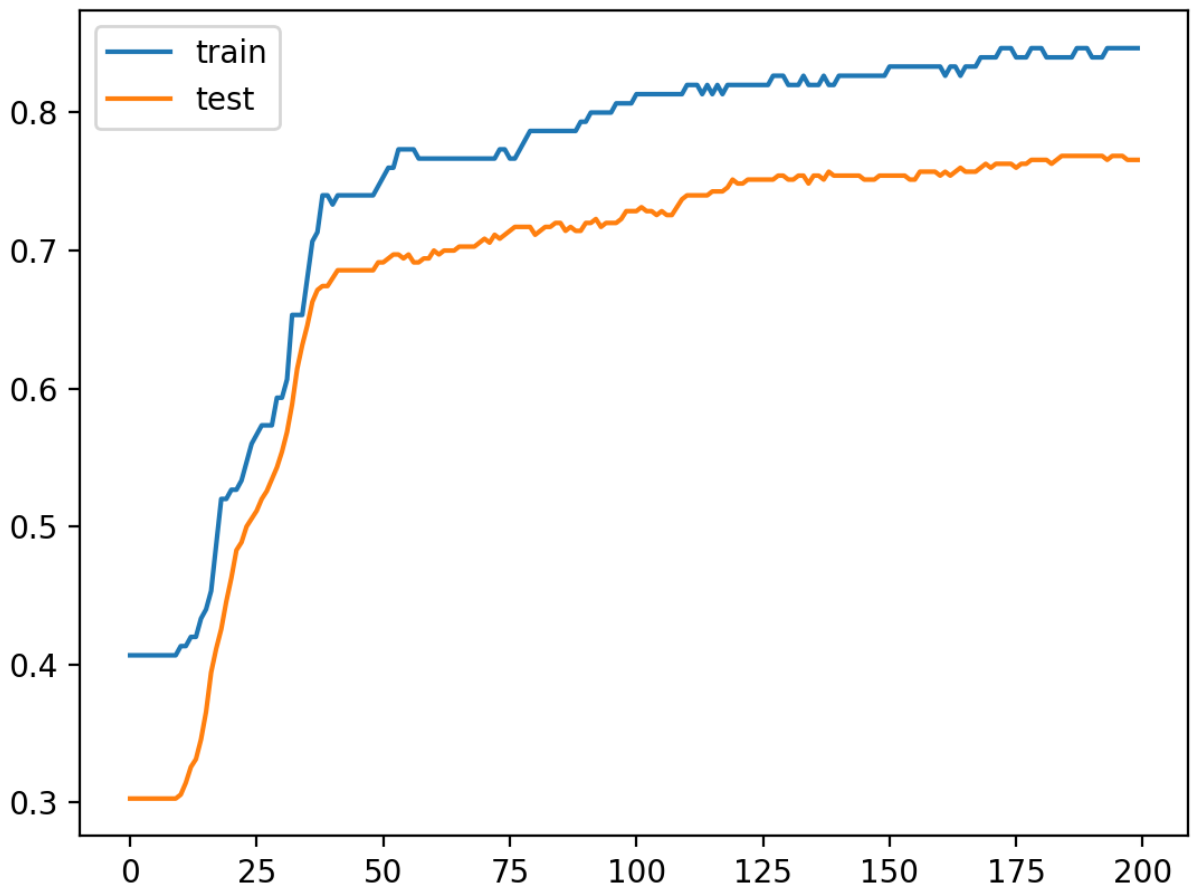
Your specific results will vary (by design!) given the high variance nature of the model.

In this case, we can see that the model achieved about 84% accuracy on the training dataset and about 76% accuracy on the test dataset; not terrible.

```
Train: 0.847, Test: 0.766
```

A line plot is also created showing the learning curves for the model accuracy on the train and test sets over each training epoch.

We can see that the model is not really overfit, but is perhaps a little underfit and may benefit from an increase in capacity, more training, and perhaps some regularization. All of these improvements of which we intentionally hold back to force the high variance for our case study.



Line Plot Learning Curves of Model Accuracy on Train and Test Dataset Over Each Training Epoch

High Variance of MLP Model

It is important to demonstrate that the model indeed has a variance in its prediction.

We can demonstrate this by repeating the fit and evaluation of the same model configuration on the same dataset and summarizing the final performance of the model.

To do this, we first split the fit and evaluation of the m
The evaluate_model() function below takes the train a
retuning the accuracy of the model on the test datase

```
# fit and evaluate a neural net model on the
def evaluate_model(trainX, trainy, testX, test
    # define model
    model = Sequential()
    model.add(Dense(15, input_dim=2, activatio
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentro
    # fit model
    model.fit(trainX, trainy, epochs=200, verb
    # evaluate the model
```

```
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    return test_acc
```

We can call this function 30 times, saving the test accuracy scores.

```
# repeated evaluation
n_repeats = 30
scores = list()
for _ in range(n_repeats):
    score = evaluate_model(trainX, trainy, testX, testy)
    print('> %.3f' % score)
    scores.append(score)
```

Once collected, we can summarize the distribution scores, first in terms of the mean and standard deviation, assuming the distribution is Gaussian, which is very reasonable.

```
# summarize the distribution of scores
print('Scores Mean: %.3f, Standard Deviation: %.3f' % (mean(scores), std(scores)))
```

We can then summarize the distribution both as a histogram to show the shape of the distribution and as a box and whisker plot to show the spread and body of the distribution.

```
# histogram of distribution
pyplot.hist(scores, bins=10)
pyplot.show()
# boxplot of distribution
pyplot.boxplot(scores)
pyplot.show()
```

The complete example of summarizing the variance of the MLP model on the chosen blobs dataset is listed below.

```
# demonstrate high variance of mlp model on blobs classification problem
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from numpy import mean
from numpy import std
from matplotlib import pyplot

# fit and evaluate a neural net model on the dataset
def evaluate_model(trainX, trainy, testX, testy):
    # define model
    model = Sequential()
    model.add(Dense(15, input_dim=2, activatio
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentro
    # fit model
    model.fit(trainX, trainy, epochs=200, verb
    # evaluate the model
    _, test_acc = model.evaluate(testX, testy,
    return test_acc

# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_
y = to_categorical(y)
# split into train and test
n_train = int(0.3 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

```
# repeated evaluation
n_repeats = 30
scores = list()
for _ in range(n_repeats):
    score = evaluate_model(trainX, trainy, testX, testy)
    print('> %.3f' % score)
    scores.append(score)
# summarize the distribution of scores
print('Scores Mean: %.3f, Standard Deviation: %.3f' % (mean(scores), std(scores)))
# histogram of distribution
pyplot.hist(scores, bins=10)
pyplot.show()
# boxplot of distribution
pyplot.boxplot(scores)
pyplot.show()
```

Running the example first prints the accuracy of each model on the test set, finishing with the mean and standard deviation of the sample of accuracy scores.

The specifics of your sample may differ, but the summary statistics should be similar.

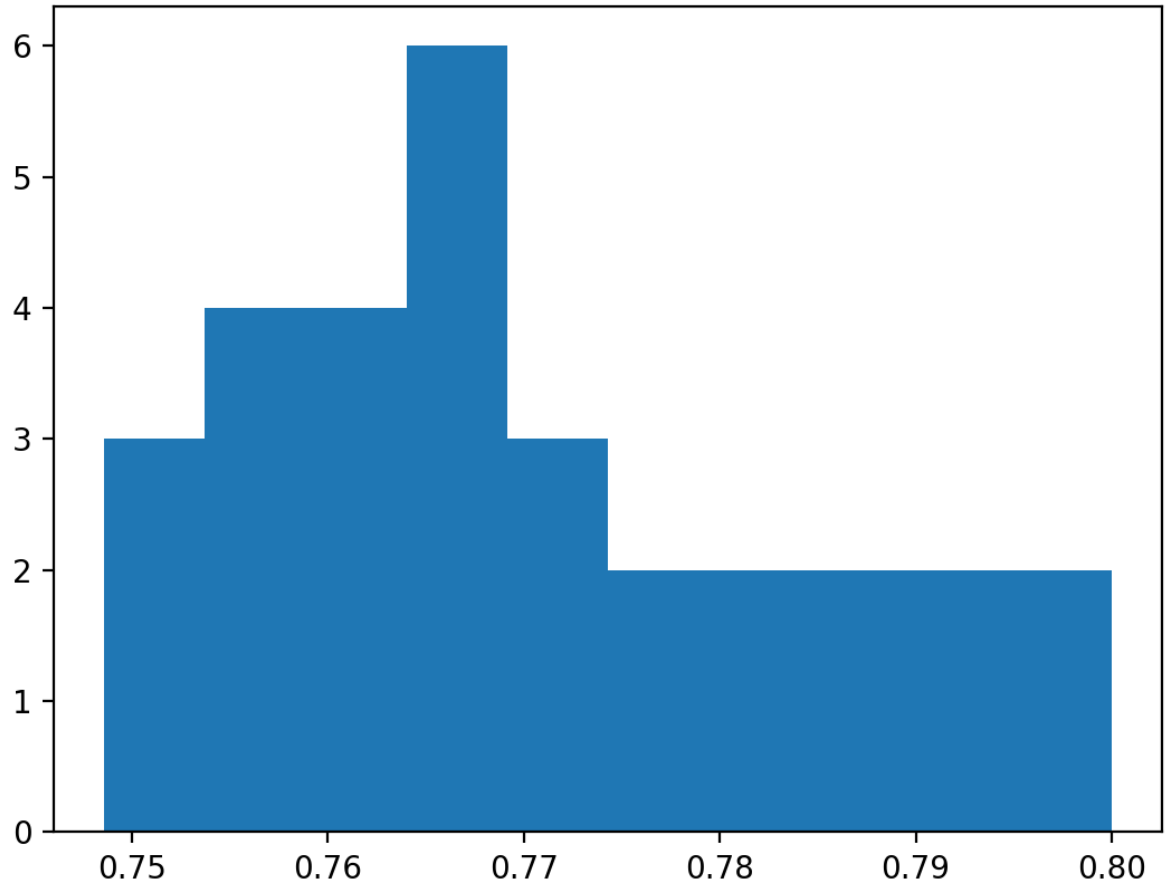
In this case, we can see that the average of the sample is 77% with a standard deviation of about 1.4%. Assuming a Gaussian distribution, we would expect 99% of accuracy scores to fall between about 73% and 81% (i.e. 3 standard deviations above and below the mean).

We can take the standard deviation of the accuracy of the model on the test set as an estimate for the variance of the predictions made by the model.

```
> 0.749
> 0.771
> 0.763
> 0.760
> 0.783
> 0.780
> 0.769
> 0.754
> 0.766
> 0.786
> 0.766
> 0.774
> 0.757
> 0.754
> 0.771
> 0.749
> 0.763
> 0.800
> 0.774
> 0.777
> 0.766
> 0.794
> 0.797
> 0.757
> 0.763
> 0.751
> 0.789
> 0.791
> 0.766
> 0.766
Scores Mean: 0.770, Standard Deviation: 0.014
```

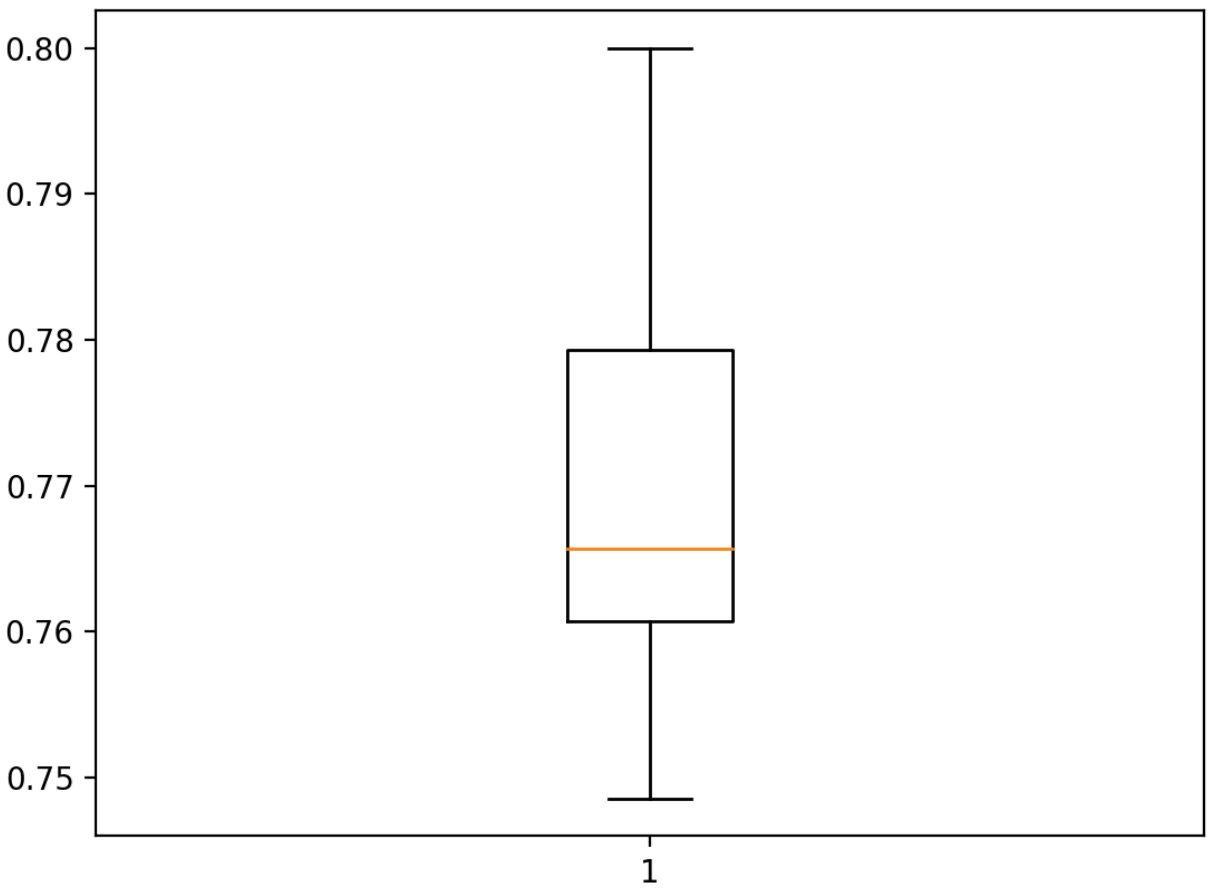
A histogram of the accuracy scores is also created, showing a very rough Gaussian shape, perhaps with a longer right tail.

A large sample and a different number of bins on the plot might better expose the true underlying shape of the distribution.



Histogram of Model Test Accuracy Over 30 Repeats

A box and whisker plot is also created showing a line at the median at about 76.5% accuracy on the test set and the interquartile range or middle 50% of the s



Box and Whisker Plot of Model Test Accuracy Over 30 Repeats

The analysis of the sample of test scores clearly demonstrates a variance in the performance of the same model trained on the same dataset.

A spread of likely scores of about 8 percentage points (81% – 73%) on the test set could reasonably be considered large, e.g. a high variance result.

Model Averaging Ensemble

We can use model averaging to both reduce the varia
generalization error of the model.

Specifically, this would result in a smaller standard de
performance on the training set. We can check both c

First, we must develop a function to prepare and retur

```
# fit model on dataset
def fit_model(trainX, trainy):
    # define model
```

```
    model = Sequential()
    model.add(Dense(15, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=200, verbose=0)
    return model
```

Next, we need a function that can take a list of ensemble members and make a prediction for an out of sample dataset. This could be one or more samples arranged in a two-dimensional array of samples and input features.

Hint: you can use this function yourself for testing ensembles and for making predictions with ensembles on new data.

```
# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result
```

We don't know how many ensemble members will be appropriate for this problem.

Therefore, we can perform a sensitivity analysis of the number of ensemble members and how it impacts test accuracy. This means we need a function that can evaluate a specified number of ensemble members and return the accuracy of a prediction combined from those members.

```
# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    print(len(subset))
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)
```

Finally, we can create a line plot of the number of ensemble members (x-axis) versus the accuracy of a prediction averaged across that many members on the test dataset (y-axis).

```
# plot score vs number of ensemble members
x_axis = [i for i in range(1, n_members+1)]
pyplot.plot(x_axis, scores)
pyplot.show()
```

The complete example is listed below.

```
# model averaging ensemble and a study of ense
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
import numpy
```

```
from numpy import array
from numpy import argmax
from sklearn.metrics import accuracy_score
from matplotlib import pyplot

# fit model on dataset
def fit_model(trainX, trainy):
    # define model
    model = Sequential()
    model.add(Dense(15, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=200, verbose=0)
    return model

# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    print(len(subset))
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = int(0.3 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
trainy = to_categorical(trainy)
# fit all models
n_members = 20
members = [fit_model(trainX, trainy) for _ in range(n_members)]
# evaluate different numbers of ensembles
scores = list()
for i in range(1, n_members+1):
    score = evaluate_n_members(members, i, tes
    print('> %.3f' % score)
    scores.append(score)
# plot score vs number of ensemble members
x_axis = [i for i in range(1, n_members+1)]
pyplot.plot(x_axis, scores)
pyplot.show()
```

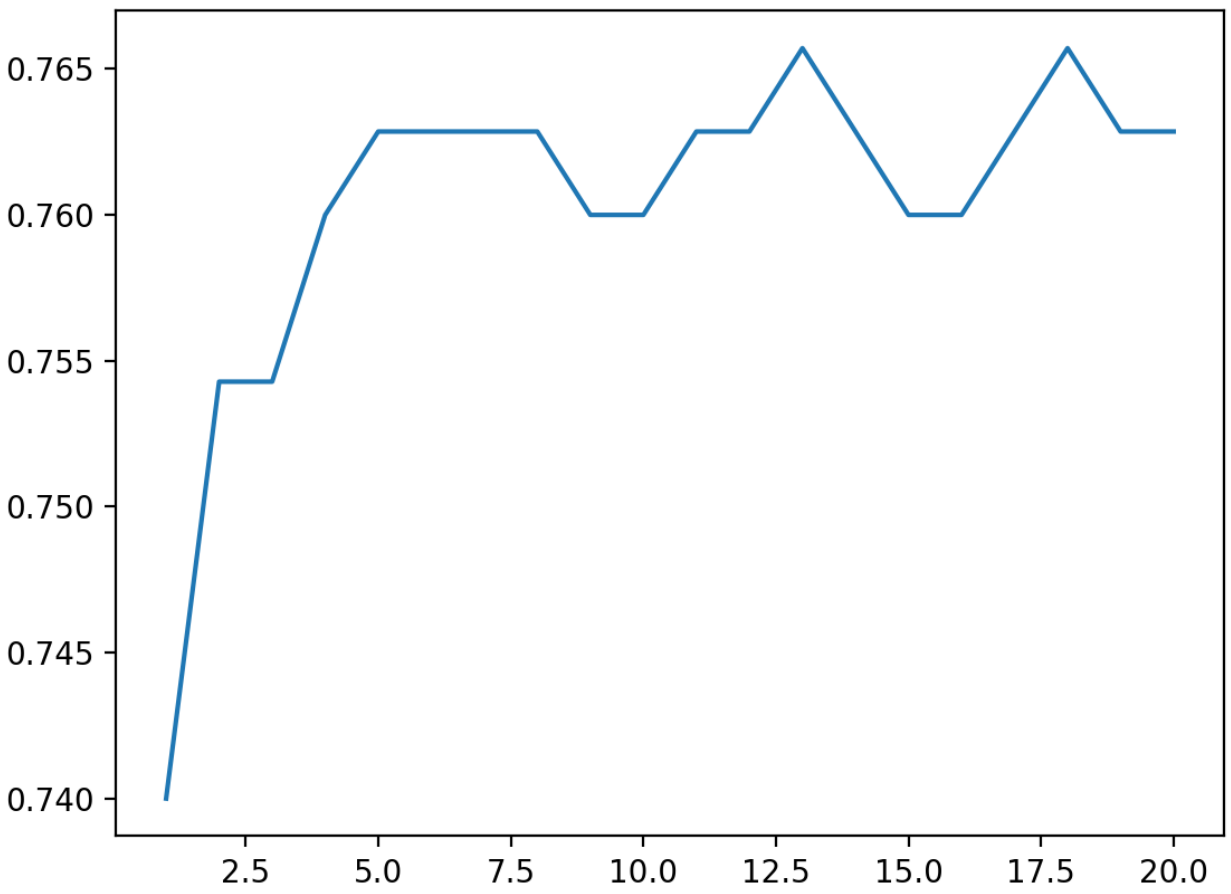
Running the example first fits 20 models on the same
on modern hardware.

Then, different sized ensembles are tested from 1 mer
printed for each ensemble size.

```
1  1
2  > 0.740
3  2
4  > 0.754
5  3
6  > 0.754
7  4
8  > 0.760
9  5
10 > 0.763
11 6
12 > 0.763
13 7
14 > 0.763
15 8
16 > 0.763
17 9
18 > 0.760
19 10
20 > 0.760
21 11
22 > 0.763
23 12
24 > 0.763
25 13
26 > 0.766
27 14
28 > 0.763
29 15
30 > 0.760
31 16
32 > 0.760
33 17
34 > 0.763
35 18
36 > 0.766
37 19
38 > 0.763
39 20
40 > 0.763
```

Finally, a line plot is created showing the relationship between ensemble size and performance on the test set.

We can see that performance improves to about five members, after which performance plateaus around 76% accuracy. This is close to the average test set performance observed during the analysis of the repeated evaluation of the model.



Line Plot of Ensemble Size Versus Model Test Accuracy

Finally, we can update the repeated evaluation experiment to use an ensemble of five models instead of a single model and compare the distribution of scores.

The complete example of a repeated evaluated five-member ensemble of the blobs dataset is listed below.

```
# repeated evaluation of model averaging ensemble on blobs dataset
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
import numpy
from numpy import array
from numpy import argmax
from numpy import mean
from numpy import std
from sklearn.metrics import accuracy_score

# fit model on dataset
def fit_model(trainX, trainy):
    # define model
    model = Sequential()
    model.add(Dense(15, input_dim=2, activati
    model.add(Dense(3, activation='softmax'))
```

```
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=200, verbose=0)
    return model

# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result

# evaluate ensemble model
def evaluate_members(members, testX, testy):
    # make prediction
    yhat = ensemble_predictions(members, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = int(0.3 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
trainy = to_categorical(trainy)
# repeated evaluation
n_repeats = 30
n_members = 5
scores = list()
for _ in range(n_repeats):
    # fit all models
    members = [fit_model(trainX, trainy) for _ in range(n_members)]
    # evaluate ensemble
    score = evaluate_members(members, testX, testy)
    print('> %.3f' % score)
    scores.append(score)
# summarize the distribution of scores
print('Scores Mean: %.3f, Standard Deviation: %.3f' % (mean(scores), std(scores)))
```

Running the example may take a few minutes as five models are fit and evaluated and this process is repeated 30 times.

The performance of each model on the test set is printed to provide an indication of progress.

The mean and standard deviation of the model perfor... fic... results may vary, but not by much.

```
> 0.769
> 0.757
> 0.754
> 0.780
> 0.771
> 0.774
> 0.766
> 0.769
> 0.774
> 0.771
```

```
> 0.760
> 0.766
> 0.766
> 0.769
> 0.766
> 0.771
> 0.763
> 0.760
> 0.771
> 0.780
> 0.769
> 0.757
> 0.769
> 0.771
> 0.771
> 0.766
> 0.763
> 0.766
> 0.771
> 0.769
Scores Mean: 0.768, Standard Deviation: 0.006
```

In this case, we can see that the average performance of a five-member ensemble on the dataset is 76%. This is very close to the average of 77% seen for a single model.

The important difference is the standard deviation shrinking from 1.4% for a single model to 0.6% with an ensemble of five models. We might expect that a given ensemble of five models on this problem to have a performance fall between about 74% and about 78% with a likelihood of 99%.

Averaging the same model trained on the same dataset gives us a spread for improved reliability, a property often highly desired in a final model to be used operationally.

More models in the ensemble will further decrease the standard deviation of the accuracy of an ensemble on the test dataset given the law of large numbers, at least to a point of diminishing returns.

This demonstrates that for this specific model and prediction problem, that a model averaging ensemble with five members is sufficient to reduce the variance of the model. This reduction in variance, in turn, also means a better on-average performance when preparing a final model.

Extensions

This section lists some ideas for extending the tutoria

- **Average Class Prediction**. Update the example the class probability prediction and compare results.
- **Save and Load Models**. Update the example to om a separate script for evaluation.
- **Sensitivity of Variance**. Create a new example th of ensemble members on the standard deviation of number of repeats and report the point of diminis

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

Further Reading

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

- Getting started with the Keras Sequential model
- Keras Core Layers API
- scipy.stats.mode API
- numpy.argmax API
- sklearn.datasets.make_blobs API

Summary

In this tutorial, you discovered how to develop a model averaging ensemble in Keras to reduce the variance in a final model.

Specifically, you learned:

- Model averaging is an ensemble learning technique that can be used to reduce the expected variance of deep learning neural network models.
- How to implement model averaging in Keras for classification and regression predictive modeling problems.
- How to work through a multi-class classification problem and use model averaging to reduce the variance of the final model.

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

Tweet Share Share

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 →

‹ Ensemble Learning Methods for Deep Learning Neural Networks

How to Create a Bagging Ensemble of Deep Learning Models in Keras ›

29 Responses to *How to Develop an Ensemble of Deep Learning Models in Keras*

bart December 22, 2018 at 2:43 am #

REPLY

Hello Jason, great post, as usual!. I am very interested in audio/sound processing and machine / deep learning application to it. Image processing is well covered in ML domain. Unfortunately audio/sound is not. I wonder if by any chance you could kindly point to some knowledge source for the topic? Or even better any introduction blog post would be much appreciated 😉

Jason Brownlee December 22, 2018 at 6:06 am #

REPLY

I hope to cover the topic in the future, thanks for the suggestion.

Sivarama Krishnan Rajaraman December 3

Great post. It helps learn a lot about ensembl
use model.save () in this example to save the ensembl
Could you please let know how to save the ensemble

Jason Brownlee December 30, 2018 at 5:41 a

Yes, see this tutorial:
https://machinelearningmastery.com/save-load-keras-deep-learning-models/

Xu Zhang September 17, 2019 at 10:43 am #

REPLY

Thank you Jason for your great post. I don't think we can use model.save() to save the ensemble model because model is not defined. We only got ensemble model predictions instead of the model itself. Your advice is highly appreciated.

Jason Brownlee September 17, 2019 at 2:34 pm #

REPLY

You can save the elements of the ensemble.

Xu Zhang September 18, 2019 at 4:20 am #

Thank you Jason. I am sorry that I can't get it.
I read your tutorial you listed. It is all about using the defined model to save the model itself or model's weights or model's architectures. I have no ideas about saving the elements of the ensemble. Are they weights? I don't think so.
When we want to output the final model for predicting new data, we have to have a saved model instead of redoing "load several saved models, ensemble together, and then get the predictions". I hope I can use model.prediction() if I have a saved ensemble model.
Could you please give us an example or a couple of lines to show how to save the elements of the ensemble? Many thanks.

Jason Brownlee September 18, 2019 at 6:19 am #

No problem.

You can call save() on each model and
make predictions.

I believe there are many examples, her
https://machinelearningmastery.com/h

Emin May 24, 2019 at 8:21 am #

Great post Jason,

I assume that there is no more point in setting seed in the beginning the code to get reproducible results, correct?

Also, I wonder, if running RandomizedSearchCV with ensemble methods is a common practice? What I mean is that in the function fit_model(params), we can write: rs = RandomizedSearchCV (…); model=rs.fit(X,y). The rest of the code is the same

Jason Brownlee May 24, 2019 at 2:27 pm #

REPLY

I don't recommend that approach as you will still get variation from input data. More here: https://machinelearningmastery.com/faq/single-faq/why-do-i-get-different-results-each-time-i-run-the-code

If you have the resources, grid searching by any means is a good idea.

Emin May 24, 2019 at 10:58 pm #

REPLY

Just to make sure I understand: setting a seed like in the link is not a good idea with ensembles. Correct?

Jason Brownlee May 25, 2019 at 7:50 am #

REPLY

Correct, in fact, I don't recommend it for neural nets in general.

Instead, it is better to mange the variance in the model and data.

HyunChul Jung June 13, 2019 at 10:45 am #

REPLY

great, i clear understand ensemble because you explains it with real code/
thanks Jason

Jason Brownlee June 13, 2019 at 2:34 pm #

Thanks.

Gabi July 27, 2019 at 8:44 pm #

thank you so much!

Jason Brownlee July 28, 2019 at 6:43 am #

REPLY

You're welcome.

Pawan August 20, 2019 at 11:10 pm #

REPLY

Hello,

When I try to use the same method as in this blog to ensemble binary models (14) to solve multiclass problem, I get the following error;
ValueError: Classification metrics can't handle a mix of multilabel-indicator and binary targets

I am loading the binary models which are already fit in my data to predict some test data. When I use that output array from the prediction to sklearn accuracy score function. I get the that error. How can I solve this? How can I join binary models for multiclass prediction much like OVA system

Jason Brownlee August 21, 2019 at 6:45 am #

REPLY

I'm not sure off hand, you will have to debug the fault.

Hira October 4, 2019 at 12:39 am #

REPLY

sir, I am a research student of bscs and my task is to ensemble three deep learning CNN pretrained models. I am using pretrained models but I can't understand how to ensemble thire results using average, majority voting and weighted average methods in Matlab . could you please help me how could I proceed.
Thanks in advance.
waiting for your guide.

Jason Brownlee October 4, 2019 at 5:45 am

Sorry, I don't have any tutorials for matlab

mah.max March 27, 2020 at 8:34 am #

Hi Jason, thank you very much for the useful tutorials. I want to combine some models, not their predicted outputs. something like concatenation, add a new layer to outputs,…..
my problem is I want a way to be able to explain how models are combined, for instance, be able to give weight to each model then combine them by averaging??
totally, could you please tell me some ways of combining models, not results?

Jason Brownlee March 28, 2020 at 6:07 am #

REPLY

The weights used to combine the predictions (e.g. weighted average) from the models might give insight into how the models are being used to create the final output.

cec May 20, 2020 at 12:29 am #

REPLY

Really nice!

One question. I have umbalaced dataset, so I did upsampling , and after that neural networks does not work really well. I tried to do an ensemble of neural network as you did, but nothing! always really low accuracy,in contrast to other models(svc ,logistic regression) which give me really high accuracy.
Could be possible that my dataset would not be trainable with a neural network?

Jason Brownlee May 20, 2020 at 6:26 am #

REPLY

Perhaps try other model types?
Perhaps try other model architectures?
Perhaps try other learning configuration?
Perhaps try other data preparation?

cec May 20, 2020 at 4:48 pm #

REPLY

1) I did
2) I did
3) I did
4) I did

☐

Jason Brownlee May 21, 2020 at 6

There are more ideas here:
https://machinelearningmastery.com/start-here/#better

Once you exhaust all of your ideas, it might be time to move on to a new project.

lorenz May 20, 2020 at 8:57 pm #

REPLY

congratulation for your tutorial!

I didn't understand why we have to make blobs.

Thanks

lorentz May 21, 2020 at 2:35 am #

REPLY

It looks like clustering learing

Jason Brownlee May 21, 2020 at 6:17 am #

REPLY

Thanks.

We don't have to. We are using the blobs dataset as the basis for exploring the models.

Leave a Reply

Name (required)

Email (will not be published) (requ

Website

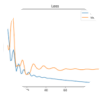
SUBMIT COMMENT

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

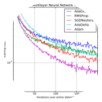
Never miss a tutorial:

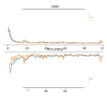
Picked for you:


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 Deep

Loving the

The Better Deep L
where I keep the *F*

SEE WHAT

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