# CHAPTER 6: Regularization and Hyperparameter Tuning

## Overview

In this chapter, you will be introduced to hyperparameter tuning. You will get hands-on experience in using TensorFlow to perform regularization on deep learning models to reduce overfitting. You will explore concepts such as L1, L2, and dropout regularization. Finally, you will look at the Keras Tuner package for performing automatic hyperparameter tuning.

By the end of the chapter, you will be able to apply regularization and tune hyperparameters in order to reduce the risk of overfitting your model and improve its performance.

## Introduction

In the previous chapter, you learned how classification models can solve problems when the response variable is discrete. You also saw different metrics used to assess the performance of such classifiers. You got hands-on experience in building and training binary, multi-class, and multi-label classifiers with TensorFlow.

When evaluating a model, you will face three different situations: model overfitting, model underfitting, and model performing. The last one is the ideal scenario, in which a model is accurately predicting the right outcome and is generalizing to unseen data well.

If a model is underfitting, it means it is neither achieving satisfactory performance nor accurately predicting the target variable. In this case, a data scientist can try tuning different hyperparameters and finding the best combination that will boost the accuracy of the model. Another possibility is to improve the input dataset by handling issues such as the cleanliness of the data or feature engineering.

A model is overfitting when it can only achieve high performance on the training set and performs poorly on the test set. In this case, the model has only learned patterns from the data relevant to the data used for training. Regularization helps to lower the risk of overfitting.

## Regularization Techniques

The main goal of a data scientist is to train a model that achieves high performance and generalizes to unseen data well. The model should be able to predict the right outcome on both data used during the training process and new data. This is the reason why a model is always assessed on the test set. This set of data serves as a proxy to evaluate the ability of the model to output correct results while in production.

Chart, scatter chart

Description automatically generated

Figure 6.1: Model not overfitting or underfitting

In Figure 6.1, the linear model (line) seems to predict relatively accurate results for both the training (circles) and test (triangles) sets.

But sometimes a model fails to generalize well and will overfit the training set. In this case, the performance of the model will be very different between the training and test sets.

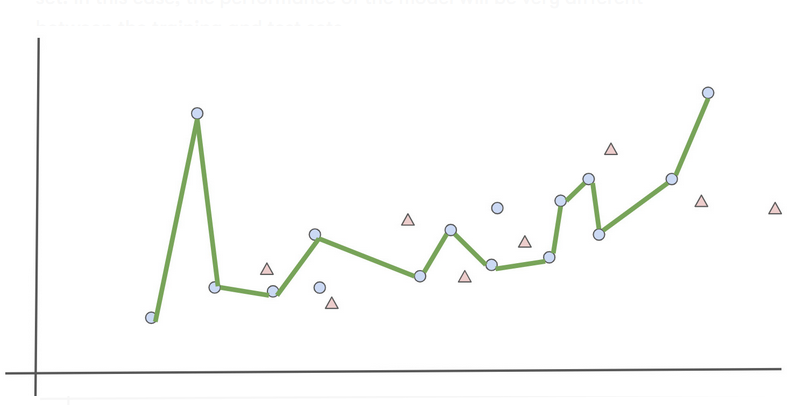


Figure 6.2: Model overfitting

Figure 6.2 shows the model (line) has only learned to predict accurately for the training set (circles) and is performing badly on the test set (triangles). This model is clearly overfitting.

Fortunately, there are **regularization techniques** that a data scientist can use to reduce and prevent overfitting, defined in the following sections.

## L1 Regularization

For deep learning models, overfitting happens when some of the features have higher weights than they should. The model puts too much emphasis on these features as it believes they are extremely important for predicting the training set. Unfortunately, these features are less relevant for the test set or any new unseen data. Regularization techniques try to penalize such weights and reduce their importance to the model predictions.

There are multiple ways to perform regularization. One of them is to add a regularization component to the cost function:

Figure 6.3: Adding a regularization component to the cost function


Figure 6.3: Adding a regularization component to the cost function

The addition of this regularization component will lead the weights of the model to be smaller as neural networks try to reduce the cost function while performing forward and backward propagations.

One very popular regularization component is L1. Its formula is as follows:

Figure 6.4: L1 regularization


Figure 6.4: L1 regularization

Formulais a hyperparameter that defines the level of penalization of the L1 regularization. W is the weight of the model. With L1 regularization, you add the sum of the absolute value of the weights to the model loss.

L1 regularization is sometimes referred to as feature selection as it tends to push the weights of non-relevant features to 0. Therefore, only the relevant features are used for making predictions.

In TensorFlow, you can define L1 regularization with the following code snippet:

*from tensorflow.keras.regularizers import l1*

*l1\_reg = l1(l=0.01)*

The l parameter corresponds to the Formula 2hyperparameter. The instantiated L1 regularization can then be added to any layer from TensorFlow Keras:

*from tensorflow.keras.layers import Dense*

*Dense(10, kernel\_regularizer=l1\_reg)*

In the preceding example, you added the L1 regularizer that you defined earlier to a fully connected layer of 10 units.

## L2 Regularization

L2 regularization is similar to L1 in that it adds a regularization component to the cost function, but its formula is different:

Figure 6.5: L2 regularization


Figure 6.5: L2 regularization

L2 regularization tends to decrease the weights of the non-relevant features. They will be close to 0, but not exactly 0. So, it reduces the impact of these features but does not disable them as L1 does.

In TensorFlow, you can define L2 regularization as follows:

*from tensorflow.keras.regularizers import l2*

*from tensorflow.keras.layers import Dense*

*l2\_reg = l2(l=0.01)*

*Dense(20, kernel\_regularizer=l2\_reg)*

In the preceding example, you defined an L2 regularizer and added it to a fully connected layer of 20 units.

TensorFlow provides another regularizer class that combines both L1 and L2 regularizers. You can instantiate it with the following code snippet:

*from tensorflow.keras.regularizers*

*import l1\_l2*

*l1\_l2\_reg = l1\_l2(l1=0.01, l2=0.001)*

In the preceding example, you instantiated L1 and L2 regularizers and specified the factors for L1 and L2 as 0.01 and 0.001, respectively. You can observe that more weights are put on the L1 regularization compared to L2. These values are hyperparameters that can be fine-tuned depending on the dataset.

In the next exercise, you will put this into practice as you apply L2 regularization to a model.

## Exercise 6.01: Predicting the radiator position of a space shuttle using L2 Regularizer

In this exercise, you will build and train a multi-class classifier in TensorFlow that will predict the radiator position of a space shuttle from eight different values using the nine different numerical features provided in this dataset.

The target variable (last column) contains seven different levels: Rad.Flow, Fpv.Close, Fpv.Open, High, Bypass, Bpv.Close, and Bpv.Open. Your goal is to accurately predict one of these seven levels using the nine features from the dataset.

The dataset can be found [here](https://raw.githubusercontent.com/fenago/tf/main/Chapter6-Regularization_and_Hyperparameter_Tuning/dataset/shuttle.trn):

1. Open a new Jupyter notebook.
2. Import the pandas library and use pd as the alias:

*import pandas as pd*

1. Create a variable called file\_url that contains the URL to the dataset:

*data\_url = "https://raw.githubusercontent.com/fenago/tf/main/Chapter6-Regularization\_and\_Hyperparameter\_Tuning/dataset/shuttle.trn"*

1. Load the dataset into a DataFrame called data using the read\_csv() function and provide the URL to the CSV file. Print the first five rows using the head() function:

*data = pd.read\_table(data\_url, header=None, sep=' ')*

*data.head()*

The expected output will be as follows:

A picture containing text

Description automatically generated

Figure 6.6: First five rows of the dataset

You can see that the dataset contains 10 columns, and they are all numeric. Also, note that the target variable (column 9) contains different class values.

1. Extract the target variable (the 9 column) using the pop() method and save it in a variable named y and save the independent variables to X:

*y = data.pop(9)*

*X = data.copy()*

1. Import train\_test\_split function from sklearn.model\_selection module

*from sklearn.model\_selection import train\_test\_split*

1. Split train and test datasets

*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30, stratify=y)*

1. Import the TensorFlow library and use tf as the alias. Then, import the Dense class from tensorflow.keras.layers:

*import tensorflow as tf*

*from tensorflow.keras.layers import Dense*

1. Set the seed as 8 to get reproducible results:

*tf.random.set\_seed(8)*

1. Instantiate a sequential model using tf.keras.Sequential() and store it in a variable called model:

*model = tf.keras.Sequential()*

1. Create a fully connected layer of 512 units with Dense() and specify ReLu as the activation function and the input shape as (9,), which corresponds to the number of features from the dataset. Save it in a variable called fc1:

*fc1 = Dense(512, input\_shape=(9,), activation='relu')*

1. Create three fully connected layers of 512, 128, and 128 units with Dense() and specify ReLu as the activation function. Save them in three variables, called fc2, fc3, and fc4, respectively:

*fc2 = Dense(512, activation='relu')*

*fc3 = Dense(128, activation='relu')*

*fc4 = Dense(128, activation='relu')*

1. Create a fully connected layer of three units (corresponding to the number of classes) with Dense() and specify softmax as the activation function. Save it in a variable called fc5:

*fc5 = Dense(8, activation='softmax')*

1. Sequentially add all five fully connected layers to the model using the add() method:

*model.add(fc1)*

*model.add(fc2)*

*model.add(fc3)*

*model.add(fc4)*

*model.add(fc5)*

1. Print the summary of the model using the summary() method:

*model.summary()*

The expected output will be as follows:

Table

Description automatically generated

Figure 6.7: Summary of the model architecture

1. Instantiate a SparseCategoricalCrossentropy() function from tf.keras.losses and save it in a variable called loss:

*loss = tf.keras.losses.SparseCategoricalCrossentropy()*

1. Instantiate Adam() from tf.keras.optimizers with 0.001 as the learning rate and save it in a variable called optimizer:

*optimizer = tf.keras.optimizers.Adam(0.001)*

1. Compile the model using the compile() method, and specify the optimizer and loss you created in *steps 16* and *17* and accuracy as the metric to be displayed:

*model.compile(optimizer=optimizer, loss=loss, \*

*metrics=['accuracy'])*

1. Start the model training process using the fit() method for five epochs and split the data into a validation set with 20% of the data:

*model.fit(X\_train, y\_train, epochs=5, validation\_split=0.2)*

The expected output will be as follows:

A picture containing table

Description automatically generated

Figure 6.8: Logs of the training process

The preceding output shows the logs of each epoch during the training of the model. Note that it took around 2 seconds to process a single epoch, and the loss value decreased from 0.8866 (first epoch) to 0.1461 (fifth epoch). Now, train another model with L2 regularization.

1. Create five fully connected layers similar to the previous model's and specify the L2 regularizer for the kernel\_regularizer parameters. Use the value 0.001 for the regularizer factor. Save the layers in five variables, called reg\_fc1, reg\_fc2, reg\_fc3, reg\_fc4, and reg\_fc5:

*reg\_fc1 = Dense(512, input\_shape=(9,), activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=0.1))*

*reg\_fc2 = Dense(512, activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=0.1))*

*reg\_fc3 = Dense(128, activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=0.1))*

*reg\_fc4 = Dense(128, activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=0.1))*

*reg\_fc5 = Dense(8, activation='softmax')*

1. Instantiate a sequential model using tf.keras.Sequential(), store it in a variable called model2, and add sequentially all five fully connected layers to the model using the add() method:

*model2 = tf.keras.Sequential()*

*model2.add(reg\_fc1)*

*model2.add(reg\_fc2)*

*model2.add(reg\_fc3)*

*model2.add(reg\_fc4)*

*model2.add(reg\_fc5)*

1. Print the summary of the model:

*model2.summary()*

The expected output will be as follows:

Table

Description automatically generated

Figure 6.9: Summary of the model architecture

1. Compile the model using the compile() method, and specify the optimizer and loss you created in *steps 16* and *17* and accuracy as the metric to be displayed:

*model2.compile(optimizer=optimizer, loss=loss, \*

*metrics=['accuracy'])*

1. Start the model training process using the fit() method for five epochs and split the data into a validation set with 20% of the data:

*model2.fit(data, target, epochs=5, validation\_split=0.2)*

The expected output will be as follows:

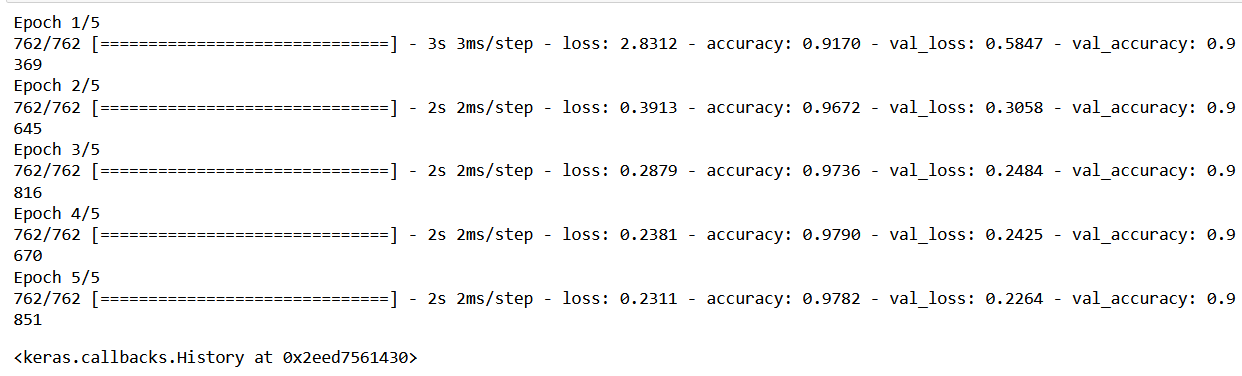


Figure 6.10: Logs of the training process

With addition of L2 regularization, the model has almost similar accuracy scores between training and test sets. The model is not overfitting.

Now that you know how to apply L1 and L2 regularization to neural networks, the next section will introduce another regularization technique, called dropout.

## Dropout Regularization

Unlike L1 and L2 regularization, dropout is a regularization technique specific to neural networks. The logic behind it is very simple: the networks will randomly change the weights of some features to 0. This will force the model to rely on other features that would have been ignored and, therefore, bump up their weights.

Diagram

Description automatically generated

Figure 6.11: Dropout of Neural networks

The preceding example shows an architecture with a dropout of 50%. This means that 50% of the units of the model are turned off at each iteration. The following code snippet shows you how to create a dropout layer of 50% in TensorFlow:

*from tensorflow.keras.layers import Dropout*

*do = Dropout(0.5)*

In the next exercise, you will extend the previous model by applying dropout.

## Exercise 6.02: Predicting the radiator position of a space shuttle using Dropout

In this exercise, you will be using the same dataset as for *Exercise 6.01*, *Predicting* the radiator position of a space shuttle from eight different values using the nine different numerical features provided in this dataset using dropout technique as regularize:

Note:

The dataset can be accessed [here](https://raw.githubusercontent.com/fenago/tf/main/Chapter6-Regularization_and_Hyperparameter_Tuning/dataset/shuttle.trn):

1. Open a new Jupyter notebook.
2. Import the pandas library and use pd as the alias:

*import pandas as pd*

1. Create a variable called file\_url that contains the URL to the dataset:

*data\_url = "https://raw.githubusercontent.com/fenago/tf/main/Chapter6-Regularization\_and\_Hyperparameter\_Tuning/dataset/shuttle.trn"*

1. Load the dataset into a DataFrame called data using the read\_csv() function and provide the URL to the CSV file. Print the first five rows using the head() function:

*data = pd.read\_table(data\_url, header=None, sep=' ')*

*data.head()*

The expected output will be as follows:

A picture containing text

Description automatically generated

Figure 6.12: First five rows of the dataset

You can see that the dataset contains 10 columns, and they are all numeric. Also, note that the target variable (column 9) contains different class values.

1. Extract the target variable (the 9 column) using the pop() method and save it in a variable named y and save the independent variables to X:

*y = data.pop(9)*

*X = data.copy()*

1. Import train\_test\_split function from sklearn.model\_selection module

*from sklearn.model\_selection import train\_test\_split*

1. Split train and test datasets

*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30, stratify=y)*

1. Import the TensorFlow library and use tf as the alias. Then, import the Dense class from tensorflow.keras.layers:

*import tensorflow as tf*

*from tensorflow.keras.layers import Dense*

*from keras.layers import Dropout*

1. Set the seed as 8 to get reproducible results:

*tf.random.set\_seed(8)*

1. Instantiate a sequential model using tf.keras.Sequential() and store it in a variable called model:

*model = tf.keras.Sequential()*

1. Create a fully connected layer of 512 units with Dense() and specify ReLu as the activation function and the input shape as (9,), which corresponds to the number of features from the dataset. Save it in a variable called fc1:

*fc1 = Dense(512, input\_shape=(9,), activation='relu')*

1. Create three fully connected layers of 512, 128, and 128 units with Dense() and specify ReLu as the activation function. Save them in three variables, called fc2, fc3, and fc4, respectively:

*fc2 = Dense(512, activation='relu')*

*fc3 = Dense(128, activation='relu')*

*fc4 = Dense(128, activation='relu')*

1. Create a fully connected layer of three units (corresponding to the number of classes) with Dense() and specify softmax as the activation function. Save it in a variable called fc5:

*fc5 = Dense(8, activation='softmax')*

1. Sequentially add all five fully connected layers to the model with a dropout layer of 0.75 in between each of them using the add() method:

*model.add(fc1)*

*model.add(Dropout(0.75))*

*model.add(fc2)*

*model.add(Dropout(0.75))*

*model.add(fc3)*

*model.add(Dropout(0.75))*

*model.add(fc4)*

*model.add(Dropout(0.75))*

*model.add(fc5)*

1. Print the summary of the model:

*model.summary()*

The expected output will be as follows:

Table

Description automatically generated

Figure 6.13: Summary of the model architecture

1. Instantiate a SparseCategoricalCrossentropy() function from tf.keras.losses and save it in a variable called loss:

*loss = tf.keras.losses.SparseCategoricalCrossentropy()*

1. Instantiate Adam() from tf.keras.optimizers with 0.001 as the learning rate and save it in a variable called optimizer:

*optimizer = tf.keras.optimizers.Adam(0.001)*

1. Compile the model using the compile() method, and specify the optimizer and loss you created in *steps 16* and *17* and accuracy as the metric to be displayed:

*model.compile(optimizer=optimizer, loss=loss, \*

*metrics=['accuracy'])*

1. Start the model training process using the fit() method for five epochs and split the data into a validation set with 20% of the data:

*model.fit(X\_train, y\_train, epochs=5, validation\_split=0.2)*

The output will be as follows:

Text

Description automatically generated

Figure 6.14: Logs of the training process

With addition of dropout, the model now has different accuracy scores between the training (0.78) and test (0.95) sets. The model is not overfitting.

You have now seen how to apply L1, L2, or dropout as regularizes for a model. In deep learning, there is another very simple technique that you can apply to avoid overfitting-that is, early stopping

## Early Stopping

Another reason why neural networks overfit is due to the training process. The more you train the model, the more it will try to improve its performance. By training the model for a longer duration (more epochs), it will at some point start finding patterns that are only relevant to the training set. In such a case, the difference between the scores of the training and test (or validation) sets will start increasing after a certain number of epochs.

To prevent this situation, you can stop the model training when the difference between the two sets starts to increase. This technique is called **early stopping**.

Diagram

Description automatically generated with low confidence

Figure 6.15: Early stopping to prevent overfitting

The preceding graph shows the loss value of a model on the training and test (or validation) sets according to the number of epochs. In early epochs, the loss value is quite different between the two sets. As the training goes on, the models start learning the relevant patterns for making predictions and both losses converge. But after a while, they start diverging. The loss of the training set keeps decreasing while the one for the test (or validation) set is increasing. You can observe that the model is overfitting and is optimizing only for the training set. Stopping the training at the point when the difference between the two losses starts to increase prevents the model from overfitting.

In TensorFlow, you can achieve this by setting up callbacks that analyze the performance of the models at each epoch and compare its score between the training and test sets. To define an early stopping callback, you will do the following:

*from tensorflow.keras.callbacks import EarlyStopping*

*EarlyStopping(monitor='val\_accuracy', patience=5)*

The preceding code shows you how to instantiate an EarlyStopping class that will monitor the accuracy score of the validation set and wait for five successive epochs with no improvement before stopping the training process.

In the next activity, you will practice applying both L1 and L2 regularization to a model.

## Activity 6.01: Predicting Motor Failure with L1 and L2 Regularizers

In this activity, you are tasked with building and training a multi-class classifier that will predict Motor Failure Time for given configuration of blades.

In this dataset, we have three independent variables pctid, x, y, and z and dependent variable wconfid

The dataset can be accessed from [here](https://raw.githubusercontent.com/fenago/tf/main/Chapter5-Classification_Models/datasets/accelerometer.csv)

1. Open a new Jupyter notebook.
2. Import the required libraries.
3. Create a data\_url and load the data using the read\_csv() method.
4. Separate the X and y variables
5. Split the data into train and test sets
6. Build the multi-class classifier with five fully connected layers of, respectively, 512, 512, 128, 128, and 26 units.
7. Train the model on the training set.

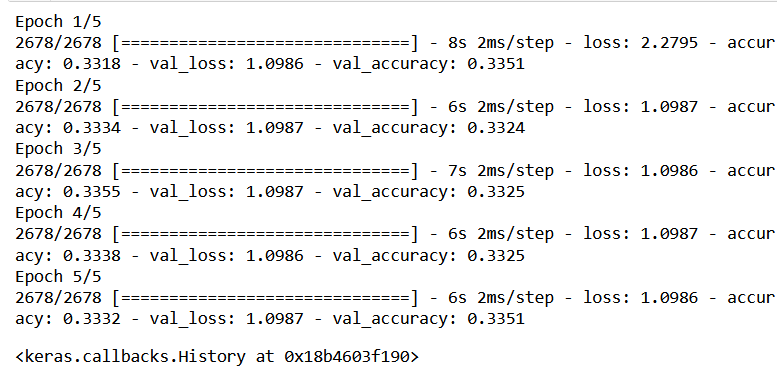


Figure 6.16: Logs of the training process

Note the solution to this activity can be found via this link

In the section ahead, you will see how to tune hyperparameters to achieve better results.

## Hyperparameter Tuning

Previously, you saw how to deal with a model that is overfitting by using different regularization techniques. These techniques help the model to better generalize to unseen data but, as you have seen, they can also lead to inferior performance and make the model underfit.

With neural networks, data scientists have access to different hyperparameters they can tune to improve the performance of a model. For example, you can try different learning rates and see whether one leads to better results, you can try different numbers of units for each hidden layer of a network, or you can test to see whether different ratios of dropout can achieve a better trade-off between overfitting and underfitting.

However, the choice of one hyperparameter can impact the effect of another one. So, as the number of hyperparameters and values you want to tune grows, the number of combinations to be tested will increase exponentially. It will also take a lot of time to train models for all these combinations—especially if you have to do it manually. There are some packages that can automatically scan the hyperparameter search space you defined and find the best combination overall for you. In the section ahead, you will see how to use one of them: Keras Tuner.

## Keras Tuner

Unfortunately, this package is not included in TensorFlow. You will need to install it manually by running the following command:

This package is very simple to use. There are two concepts to understand: **hyperparameters** and **tuners**.

Hyperparameters are the classes used to define a parameter that will be assessed by the tuner. You can use different types of hyperparameters. The main ones are the following:

* hp.Boolean: A choice between True and False
* hp.Int: A choice with a range of integers
* hp.Float: A choice with a range of decimals
* hp.Choice: A choice within a list of possible values

The following code snippet shows you how to define a hyperparameter called learning\_rate that can only take one of four values—0.1, 0.01, 0.001, or 0.0001:

*hp.Choice('learning\_rate', values = [0.1, 0.01, 0.001, 0.0001])*

A tuner in the Keras Tuner package is an algorithm that will look at the hyperparameter search space, test some combinations, and find the one that gives the best result. The Keras Tuner package provides different tuners, and in the section ahead, you will look at three of them: **random search**, **Hyperband**, and **Bayesian optimization**.

Once defined with the algorithm of your choice, you can call the search() method to start the hyperparameter tuning process on the training and test sets, as follows:

*tuner.search(X\_train, y\_train, validation\_data=(X\_test, y\_test))*

Once the search is complete, you can access the best combination with

get\_best\_hyperparameters()

and then look specifically at one of the hyperparameters you defined:

*best\_hps = tuner.get\_best\_hyperparameters()[0]*

*best\_hps.get('learning\_rate')*

Finally, the hypermodel.build() method will instantiate a TensorFlow Keras model with the best hyperparameters found:

*model = tuner.hypermodel.build(best\_hps)*

It's as simple as that. Now, let's have a look at the random search tuner.

## Random Search

Random search is one of the available algorithms in this package. As its name implies, it randomly defines the combinations to be tested by sampling through the search space. Even though this algorithm doesn't test every single possible combination, random search provides very good results.

Note

The algorithm that tests every single combination of the search space is called grid search.

Chart, scatter chart

Description automatically generated

Figure 6.17: Comparison between grid search and random search

The preceding figure shows an example of the difference between grid search and random search. You can see that grid search splits the search space into a grid and tests each of the combinations, but some may lead to the same loss value, which makes it less efficient. On the other side, random search covers the search space more efficiently and helps find the optimal solution.

In Keras Tuner, before instantiating a tuner, you need to define a model-building function that will define the architecture of the TensorFlow Keras model to be trained with the hyperparameters you want to test. Here is an example of such a function:

*def model\_builder(hp):*

*model = tf.keras.Sequential()*

*hp\_lr = hp.Choice('learning\_rate', \*

*values = [0.1, 0.01, 0.001, 0.0001])*

*model.add(Dense(512, input\_shape=(100,), activation='relu'))*

*model.add(Dense(128, activation='relu'))*

*model.add(Dense(10, activation='softmax'))*

*loss = tf.keras.losses.SparseCategoricalCrossentropy()*

*optimizer = tf.keras.optimizers.Adam(hp\_lr)*

*model.compile(optimizer=optimizer, loss=loss, \*

*metrics=['accuracy'])*

*return model*

In the preceding code snippet, you created a model composed of three fully connected layers of 512, 128, and 10 units that will be trained with a categorical cross-entropy loss function and the Adam optimizer. You defined the learning\_rate hyperparameter that will be assessed by Keras Tuner.

Once the model-building function is defined, you can instantiate a random search tuner like the following:

*import kerastuner as kt*

*tuner = kt.RandomSearch(model\_builder, objective='val\_accuracy', \*

*max\_trials=10)*

In the preceding code, you instantiated a RandomSearch tuner that will look at the model and hyperparameters defined in the model\_builder function using the validation accuracy as the objective metric and will run for a maximum of 10 trials.

In the next exercise, you will use random search to find the best set of hyperparameters for a model.

## Exercise 6.03: Predicting the radiator position of a space shuttle using Random Search from Keras Tuner

In this exercise, you will be using the same dataset as for *Exercise 6.01*, *Predicting the radiator position of a space shuttle using the L2 Regularizer*. You will build and train a multi-class model in TensorFlow that will predict the class using the Keras Tuner package to find the best regularization factor for L2 regularization through random search:

Note

The dataset can be found [here](https://raw.githubusercontent.com/fenago/tf/main/Chapter6-Regularization_and_Hyperparameter_Tuning/dataset/shuttle.trn):

1. Open a new Jupyter notebook.
2. Import the pandas library and use pd as the alias:

*import pandas as pd*

1. Create a variable called file\_url that contains the URL to the dataset:

*data\_url = "https://raw.githubusercontent.com/fenago/tf/main/Chapter6-Regularization\_and\_Hyperparameter\_Tuning/dataset/shuttle.trn"*

1. Load the dataset into a DataFrame called data using the read\_csv() function and provide the URL to the CSV file. Print the first five rows using the head() function:

*data = pd.read\_table(data\_url, header=None, sep=' ')*

*data.head()*

The expected output will be as follows:

A picture containing text

Description automatically generated

Figure 6.12: First five rows of the dataset

You can see that the dataset contains 10 columns, and they are all numeric. Also, note that the target variable (column 9) contains different class values.

1. Extract the target variable (the 9 column) using the pop() method and save it in a variable named y and save the independent variables to X:

*y = data.pop(9)*

*X = data.copy()*

1. Import train\_test\_split function from sklearn.model\_selection module

*from sklearn.model\_selection import train\_test\_split*

1. Split train and test datasets

*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30, stratify=y)*

1. Install the kerastuner package and then import it and assign it the kt alias:

*!pip install keras-tuner*

*import keras\_tuner as kt*

1. Import the TensorFlow library and use tf as the alias. Then, import the Dense class from tensorflow.keras.layers:

*import tensorflow as tf*

*from tensorflow.keras.layers import Dense*

1. Set the seed as 8 using tf.random.set\_seed() to get reproducible results:

*tf.random.set\_seed(8)*

1. Define a function called model\_builder that will create a sequential model with the same architecture as Exercise 6.02, Predicting the space shuttle Outcome Using Dropout, with L2 regularization, but this time, provide an hp.Choice hyperparameter for the regularization factor:

*def model\_builder(hp):*

*model = tf.keras.Sequential()*

*hp\_l2 = hp.Choice('l2', values = [0.1, 0.01, 0.001, 0.0001])*

*reg\_fc1 = Dense(512, input\_shape=(9,), activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=hp\_l2))*

*reg\_fc2 = Dense(512, activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=hp\_l2))*

*reg\_fc3 = Dense(128, activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=hp\_l2))*

*reg\_fc4 = Dense(128, activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=hp\_l2))*

*reg\_fc5 = Dense(8, activation='softmax')*

*model.add(reg\_fc1)*

*model.add(reg\_fc2)*

*model.add(reg\_fc3)*

*model.add(reg\_fc4)*

*model.add(reg\_fc5)*

*loss = tf.keras.losses.SparseCategoricalCrossentropy()*

*optimizer = tf.keras.optimizers.Adam(0.001)*

*model.compile(optimizer = optimizer, loss = loss, \*

*metrics = ['accuracy'])*

*return model*

1. Instantiate a RandomSearch tuner and assign val\_accuracy to objective and 10 to max\_trials:

*tuner = kt.RandomSearch(model\_builder, objective='val\_accuracy', \*

*max\_trials=10)*

1. Launch the hyperparameter search with the search() method on the training and test sets:

*tuner = kt.RandomSearch(model\_builder, objective='val\_accuracy', \*

*max\_trials=10)*

1. Extract the best hyperparameter combination (index 0) with get\_best\_hyperparameters() and save it in a variable called best\_hps:

*tuner.search(X\_train, y\_train, validation\_data=(X\_test, y\_test))*

1. Extract the best hyperparameter combination (index 0) with

get\_best\_hyperparameters() and save it in a variable called best\_hps:

*best\_hps = tuner.get\_best\_hyperparameters()[0]*

1. Extract the best value for the l2 regularization hyperparameter, save it in a variable called best\_l2, and print its value:

*best\_hps = tuner.get\_best\_hyperparameters()[0]*

You should get the following result:

The best value for the l2 hyperparameter found by random search is 0.0001.



1. Start the model training process using the fit() method for five epochs and use the test set for validation\_data:

*model = tuner.hypermodel.build(best\_hps)*

*model.fit(X\_train, y\_train, epochs=5, \*

*validation\_data=(X\_test, y\_test))*

You should get the following output:

Table

Description automatically generated

Figure 6.19: Logs of the training process

Using a random search tuner, you found the best value for L2 regularization (0.0001), which helped the model to achieve an accuracy of 0.83 on the training set and 0.81 on the test set. These scores are quite an improvement on those from *Exercise 6.01*, *Predicting the space shuttle Outcome Using the L2 Regularizer* (0.69 for the training set and 0.59 for the test set).

In the next section, you will use another Keras tuner, called Hyperband.

## Hyperband

Hyperband is another tuner available in the Keras Tuner package. Like random search, it randomly picks candidates from the search space, but more efficiently. The idea behind it is to test a set of combinations for just one or two iterations, keeping only the best performers and training them for longer. So, the algorithm doesn't waste time in training non-performing combinations as with random search. Instead, it simply discards them from the next run. Only the ones that achieve higher performance are kept for longer training. To instantiate a Hyperband tuner, execute the following command:

*tuner = kt.Hyperband(model\_builder, objective='val\_accuracy', \*

*max\_epochs=5)*

This tuner takes a model-building function and an objective metric as input parameters, as for random search. But it requires an additional one, max\_epochs, corresponding to the maximum number of epochs a model is allowed to train for during the hyperparameter search.

## Exercise 6.04: Predicting the radiator position of a space shuttle using Hyperband from Keras Tuner

In this exercise, you will be using the same dataset as for *Exercise 6.01*, *Predicting the radiator position of a space shuttle using the L2 Regularizer*. You will build and train a multi-class model in TensorFlow that will predict the class outcome using the Keras Tuner package to find the best learning rate and the number of units for the input layer through random search:

Note

The dataset can be found [here](https://raw.githubusercontent.com/fenago/tf/main/Chapter6-Regularization_and_Hyperparameter_Tuning/dataset/shuttle.trn):

1. Open a new Jupyter notebook.
2. Import the pandas library and use pd as the alias:

*import pandas as pd*

1. Create a variable called file\_url that contains the URL to the dataset:

*data\_url = "https://raw.githubusercontent.com/fenago/tf/main/Chapter6-Regularization\_and\_Hyperparameter\_Tuning/dataset/shuttle.trn"*

1. Load the dataset into a DataFrame called data using the read\_csv() function and provide the URL to the CSV file. Print the first five rows using the head() function:

*data = pd.read\_table(data\_url, header=None, sep=' ')*

*data.head()*

The expected output will be as follows:

A picture containing text

Description automatically generated

Figure 6.12: First five rows of the dataset

You can see that the dataset contains 10 columns, and they are all numeric. Also, note that the target variable (column 9) contains different class values.

1. Extract the target variable (the 9 column) using the pop() method and save it in a variable named y and save the independent variables to X:

*y = data.pop(9)*

*X = data.copy()*

1. Import train\_test\_split function from sklearn.model\_selection module

*from sklearn.model\_selection import train\_test\_split*

1. Split train and test datasets

*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30, stratify=y)*

1. Install the kerastuner package and then import it and assign it the kt alias:

*!pip install keras-tuner*

*import keras\_tuner as kt*

1. Import the TensorFlow library and use tf as the alias. Then, import the Dense class from tensorflow.keras.layers:

*import tensorflow as tf*

*from tensorflow.keras.layers import Dense*

1. Set the seed as 8 using tf.random.set\_seed() to get reproducible results:

*tf.random.set\_seed(8)*

1. Define a function called model\_builder that will create a sequential model with the same architecture as Exercise 6.02, Predicting the space shuttle Outcome Using Dropout, with L2 regularization and a 0.0001 regularization factor, but this time, provide a huperparameter, hp.Choice, for the learning rate (0.01, 0.001, or 0.0001) and hp.Int function for the number of units (between 128 and 512 with a step of 64) for this input fully connected layer:

*def model\_builder(hp):*

*model = tf.keras.Sequential()*

*hp\_units = hp.Int('units', min\_value=128, max\_value=512, \*

*step=64)*

*reg\_fc1 = Dense(hp\_units, input\_shape=(42,), \*

*activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=0.0001))*

*reg\_fc2 = Dense(512, activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=0.0001))*

*reg\_fc3 = Dense(128, activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=0.0001))*

*reg\_fc4 = Dense(128, activation='relu', \*

*kernel\_regularizer=tf.keras.regularizers\*

*.l2(l=0.0001))*

*reg\_fc5 = Dense(3, activation='softmax')*

*model.add(reg\_fc1)*

*model.add(reg\_fc2)*

*model.add(reg\_fc3)*

*model.add(reg\_fc4)*

*model.add(reg\_fc5)*

*loss = tf.keras.losses.SparseCategoricalCrossentropy()*

*hp\_learning\_rate = hp.Choice('learning\_rate', \*

*values = [0.01, 0.001, 0.0001])*

*optimizer = tf.keras.optimizers.Adam(hp\_learning\_rate)*

*model.compile(optimizer = optimizer, loss = loss, \*

*metrics = ['accuracy'])*

*return model*

1. Instantiate a Hyperband tuner and assign val\_accuracy to objective and 5 to max\_epochs:

*tuner = kt.Hyperband(model\_builder, objective='val\_accuracy', \*

*max\_epochs=5)*

1. Launch the hyperparameter search with search() on the training and test sets:

*tuner = kt.Hyperband(model\_builder, objective='val\_accuracy', \*

*max\_epochs=5)*

1. Extract the best hyperparameter combination (index 0) with

get\_best\_hyperparameters()

and save it in a variable called best\_hps:

*best\_hps = tuner.get\_best\_hyperparameters()[0]*

1. Extract the best value for the number of units for the input layer, save it in a variable called best\_units, and print its value:

*best\_units = best\_hps.get('units')*

*best\_units*

You will get the following output:



1. Extract the best value for the learning rate, save it in a variable called best\_lr, and print its value:

*best\_lr = best\_hps.get('learning\_rate')*

*best\_lr*

The output will be the following:



1. Start the model training process using the fit() method for five epochs and use the test set for validation\_data:

*model = tuner.hypermodel.build(best\_hps)*

*model.fit(X\_train, y\_train, epochs=5, validation\_data=(X\_test, y\_test))*

You will get the following output:

Text, table

Description automatically generated with medium confidence

Figure 6.21: Logs of the training process

Using Hyperband as the tuner, you found the best number of units for the input layer (192) and learning rate (0.001). With these hyperparameters, the final model achieved an accuracy of 0.81 on both the training and test sets. It is not overfitting much and achieved a satisfactory accuracy score.

Another very popular tuner is Bayesian optimization, which you will learn about in the following section.

## Bayesian Optimization

Bayesian optimization is another very popular algorithm used for automatic hyperparameter tuning. It uses probabilities to determine the best combination of hyperparameters. The objective is to iteratively build a probability model that optimizes the objective function from a set of hyperparameters. At each iteration, the probability model is updated from the results obtained. Therefore, unlike random search and Hyperband, Bayesian optimization takes past results into account to improve new ones. The following code snippet will show you how to instantiate a Bayesian optimizer in Keras Tuner:

*tuner = kt.BayesianOptimization(model\_builder, objective='val\_accuracy', \*

*max\_trials=10)*

The expected parameters are similar to random search, including the model-building function, the objective metric, and the maximum number of trials.

In the following activity, you will use Bayesian optimization to predict the income of a person.

## Activity 6.02: Predicting Motor Failure with Bayesian Optimization from Keras Tuner

In this activity, you are tasked with building and training a multi-class classifier that will predict Motor Failure Time for given configuration of blades.

In this dataset, we have three independent variables pctid, x, y, and z and dependent variable wconfid

You will perform automatic hyperparameter tuning with keras Tuner and find the best combination of hyperparameters for the learning rate, the number of units for the input layer, and L2 regularization with Bayesian optimization.

The dataset can be accessed from [here](https://raw.githubusercontent.com/fenago/tf/main/Chapter5-Classification_Models/datasets/accelerometer.csv)

1. Open a new Jupyter notebook.
2. Import the required libraries.
3. Create a data\_url and load the data using the read\_csv() method.
4. Separate the X and y variables
5. Split the data into train and test sets
6. Build the multi-class classifier with five fully connected layers of, respectively, 512, 512, 128, 128, and 26 units and the three different hyperparameters to be tuned: the learning rate (between 0.01 and 0.001), the number of units for the input layer (between 128 and 512 and a step of 64), and L2 regularization (between 0.1, 0.01, and 0.001).
7. Find the best combination of hyperparameters with Bayesian optimization
8. Train the model on the training set with the best hyper parameters found.

The expected output will be as follows:

Text

Description automatically generated

Figure 6.22: Logs of the training process

Note

The solution to this activity can be found via this link.

## Summary

You started your journey in this chapter with an introduction to the different scenarios of training a model. A model is overfitting when its performance is much better on the training set than the test set. An underfitting model is one that can achieve good results only after training. Finally, a good model achieves good performance on both the training and test sets.

Then, you encountered several regularization techniques that can help prevent a model from overfitting. You first looked at the L1 and L2 regularizations, which add a penalty component to the cost function. This additional penalty helps to simplify the model by reducing the weights of some features. Then, you went through two different techniques specific to neural networks: dropout and early stopping. Dropout randomly drops some units in the model architecture and forces it to consider other features to make predictions. Early stopping is a mechanism that automatically stops the training of a model once the performance of the test set starts to deteriorate.

After this, you learned how to use the Keras Tuner package for automatic hyperparameter tuning. You considered three specific types of tuners: random search, Hyperband, and Bayesian optimization. You saw how to instantiate them, perform a hyperparameter search, and extract the best values and model. This process helped you to achieve better performance on the models trained for the exercises and activities.

In the next chapter, you will learn more about **Convolutional Neural Networks** (**CNNs**). Such architecture has led to groundbreaking results in computer vision in the past few years. The following chapter will show you how to use this architecture to recognize objects in images.