CHAPTER 5

Tuning and Deploying Deep Neural Networks

So far in the journey of this book, we have primarily talked about how to develop a DNN for a given use case and looked at a few strategies and rules of thumb to bypass roadblocks we could face in the process. In this chapter, we will discuss the journey onward after developing the initial model by exploring the methods and approaches you need to implement when the model developed doesn't perform to your expectations. We will discuss regularization and hyperparameter tuning, and toward the end of the chapter, we will also have a high-level view of the process to deploy a model after tuning. However, we won't actually discuss the implementation specifics of deploying; this will just be an overview offering guidelines to achieve success in the process. Let's get started.

The Problem of Overfitting

In the process of developing and training ML and DL models, you will often come across a scenario where the trained model seems to perform well on the training dataset but fails to perform similarly on the test dataset. In data science, this phenomenon is called "overfitting." In a literal sense, your model overfits the data. Although you have crossed paths with this term previously in this book, we haven't discussed this topic in detail so far. Let's try to understand this phenomenon in a more simplified way.

J. Moolayil, *Learn Keras for Deep Neural Networks*,
https://doi.org/10.1007/978-1-4842-4240-7_5

The process of training a model is called "fitting the data"; the neural network learns the latent patterns within the data and mathematically improves the model weights/structure to suit the patterns it discovers in the learning process. In a nutshell, training the model adapts the its structure (weights) to fit the data (patterns) and thereby improves its performance. This beautiful process gets complicated at the point when the pattern it discovers turns out to be merely noise in reality. Unfortunately, the mathematical equation doesn't have the prowess to always distinguish between a signal and a noise (by noise, we mean a data point that doesn't represent the training sample but comes about due to random chance). When it fails, it learns the noise too and adjusts its weight to accommodate the new signal, which was noise in reality.

To understand this process, let's take a simple example. Say a five-year-old loves to eat cakes baked by his mother. He demands cakes to be baked every day at home. His mother politely denies these demands, but assures him that she will bake cakes on certain occasions. The little boy now looks forward to each new day, hoping that it will be one of those occasions when his mother will bake a cake. His mother, on the other hand, had no real intention to find occasions to bake cakes. She would simply bake a cake every Sunday when she had time off from work. The five-year-old continues to watch every day and slowly learns that his mom will bake a cake on every Sunday. So, he learns the following pattern: "If day == Sunday, then Mother will bake cakes." One fine Sunday, his mother had to travel for an errand and was left with no time to bake a cake. The five-year-old couldn't understand his pattern breaking down. So, to accommodate the new event, he modified his rules by formulating the new pattern as follows: "If day == Sunday, then Mother will bake a cake, but if the day is in the last week of the month, then no cake." In reality, the Sunday his mother missed baking the cake was a noise. He should have ideally ignored that and kept his previously learned pattern intact. But unfortunately, he failed to distinguish between signal and noise and thereby overcomplicated his rules and overfit the data.

Similarly, when a DL model learns from the noise and accommodates by adjusting the weights to suit the noise, it overfits the data. This problem becomes serious, since learning noise results in a significant decrease in model performance. That is the reason you would observe a large gap between the performance of a model on training data and the performance on unseen data. Circumventing this problem and tailoring a model's learning process to accommodate only signals (or real patterns) instead of noise can be achieved to a great extent (though not fully) with regularization.

So, What Is Regularization?

In simplest terms, regularization is a process to reduce overfitting. It is a mathematical way of inducing a warning into the model's learning process when it accommodates noise. To give a more realistic definition, it is a method to penalize the model weights in the event of overfitting.

Let's understand this process in a very simple way. In DL, the weights of the neuron connections are updated after each iteration. When the model encounters a noisy sample and assumes the sample is a valid one, it tries to accommodate the pattern by updating the weights aligned with the noise. In realistic data samples, noisy data points don't resemble anything close to a regular data point; they are far off from them. So, the weight updates will also be in sync with the noise (i.e., the change in weight will be huge). The process of regularization adds the weights of the edges to the defined loss function and holistically represents a higher loss. The network then tunes itself to reduce the loss and thereby makes the weight updates in the right direction; this works by ignoring the noise rather than accommodating it in the learning process.

The process of regularization can be demonstrated as

$$\textit{Cost Function} = \textbf{\textit{Loss}}\ \textit{(as defined for the model)} + \textbf{\textit{Hyperparameter}} \times \textbf{\textit{[Weights]}}$$

The hyperparameter is represented as $\frac{\lambda}{2m}$ and the value of λ is defined by the user.

Based on how the weights are added to the loss function, we have two different types of regularization techniques: L1 and L2.

L1 Regularization

In L1 regularization, the absolute weights are added to the loss function. To make the model more generalized, the values of the weights are reduced to 0, and therefore this method is strongly preferred when we are trying to compress the model for faster computation.

The equation can be represented as

$$\text{Cost Function} = \text{Loss (as defined)} + \frac{\lambda}{2m} * \sum \|Weights\|$$

In Keras, the L1 loss can be added to a layer by providing the 'regularizer' object to the 'kernel regularizer' parameter. The following code snippet demonstrates adding an L1 regularizer to a dense layer in Keras.

```
from keras import regularizers
from keras import Sequential

model = Sequential()
model.add(Dense(256, input_dim=128,
kernel_regularizer=regularizers.l1(0.01))
```

The value of 0.01 is the hyperparameter value we set for λ.

L2 Regularization

In L2 regularization, the squared weights are added to the loss function. To make the model more generalized, the values of the weights are reduced to near 0 (but not actually 0), and hence this is also called the "weight decay" method. In most cases, L2 is highly recommended over L1 for reducing overfitting.

The equation can be represented as

$$\text{Cost Function} = \text{Loss (as defined)} + \frac{\lambda}{2m} * \|Weights\|^2$$

We can add an L2 regularizer to a DL model just like L1. The following code snippet demonstrates adding an L2 regularizer to the dense layer.

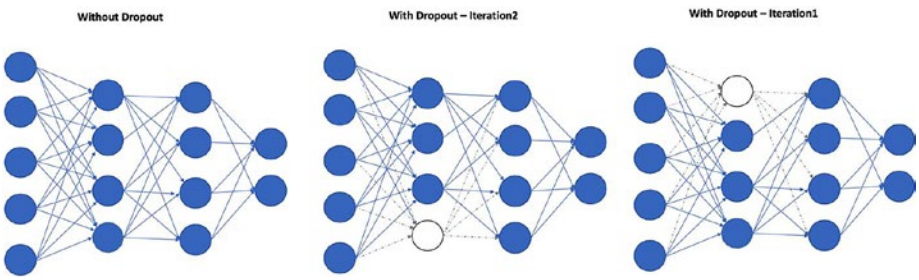
```
model = Sequential()
model.add(Dense(256, input_dim=128,
kernel_regularizer=regularizers.l2(0.01))
```

The value of 0.01 is the hyperparameter value we set for λ.

Dropout Regularization

In addition to L1 and L2 regularization, there is another popular technique in DL to reduce overfitting. This technique is to use a dropout mechanism. In this method, the model arbitrarily drops or deactivates a few neurons for a layer during each iteration. Therefore, in each iteration the model looks at a slightly different structure of itself to optimize (as a couple of neurons and the connections would be deactivated). Say we have two successive layers, H1 and H2, with 15 and 20 neurons, respectively. Applying the dropout technique between these two layers would result in randomly dropping a few neurons (based on a defined percentage) for H1, which therefore reduces the connections between H1 and H2. This process repeats for each iteration with randomness, so if the model has learned for a batch and updated the weights, the next batch might have a fairly different set of weights and connections to train. The process is not only efficient due to the reduced computation but also works intuitively in reducing the overfitting and therefore improving the overall performance.

The idea of dropout can be visually understood using the following figure. We can see that the regular network has all neurons and connections between two successive layers intact. With dropout, each iteration induces a certain defined degree of randomness by arbitrarily deactivating or dropping a few neurons and their associated weight connections.



In Keras, we can use dropout to a layer with the following convention:

```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

The following code snippet showcases dropout added to the dense hidden layer. The parameter value of 0.25 indicates the dropout rate (i.e., the percentage of the neurons to be dropped).

```
from keras import Sequential
from keras.layers.core import Dropout, Dense

model = Sequential()
model.add(Dense(100, input_dim= 50, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(1,activation="linear"))
```

Hyperparameter Tuning

Hyperparameters are the parameters that define a model's holistic structure and thus the learning process. We can also relate hyperparameters as the metaparameter for a model. It differs from a model's actual parameters, which it learns during the training process (say, the model weights). Unlike model parameters, hyperparameters cannot be learned; we need to tune them with different approaches to get improved performance.

To understand this topic better, let us look at the definition in a more simplified way. When we design a DNN, the architecture of the model is defined by a few high-level artifacts. These artifacts could be the number of neurons in a layer, the number of hidden layers, the activation function, the optimizer, the learning rate of the architecture, the number of epochs, batch size, and so on. All of these parameters are collectively used to design a network, and they have a huge impact on the model's learning process and its end performance. These parameters cannot be trained; in fact, they need to be selected with experience and judgment, just like the rules we learned in Chapter 3 to decide the size of the architecture to start with. Parameters that define the model's holistic architecture overall are collectively called hyperparameters. Choosing the right hyperparameters is an intensive and iterative process, but it becomes easier with experience. The process of experimenting with different values for hyperparameters to improve the overall model process is called model tuning or hyperparameter tuning.

Hyperparameters in DL

Let's have a look at the different hyperparameters available for a DL model and study the available options to choose from. We will then look at various approaches for selecting the right set of hyperparameters for a model.

Number of Neurons in a Layer

For most classification and regression use cases using tabular cross-sectional data, DNNs can be made robust by playing around with the width of the network (i.e., the number of neurons in a layer). Generally, a simple rule of thumb for selecting the number of neurons in the first layer is to refer to the number of input dimensions. If the final number of input dimensions in a given training dataset (this includes the one-hot

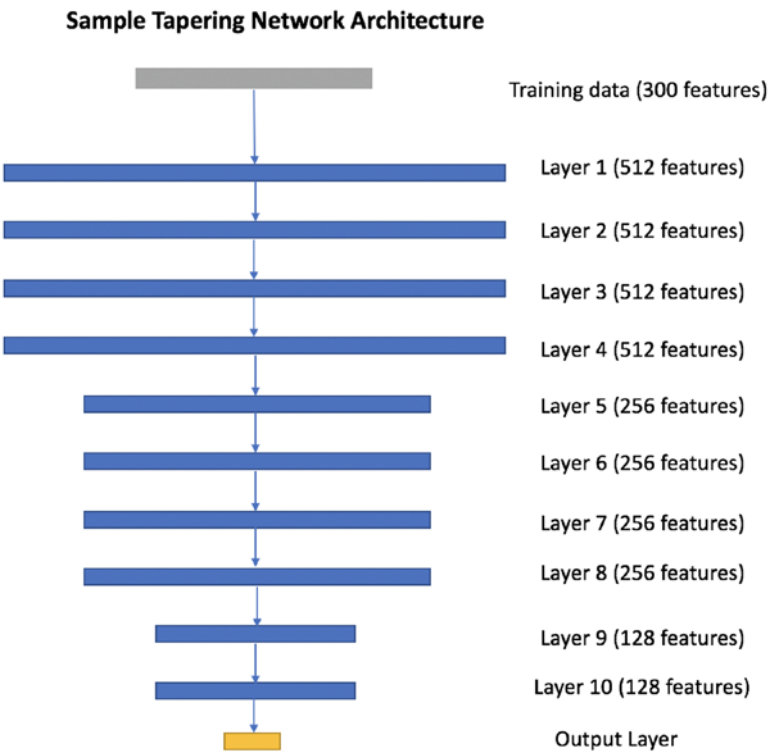
encoded features also) is x, we should use at least the closest number to 2x in the power of 2. Let's say you have 100 input dimensions in your training dataset: preferably start with 2 × 100 = 200, and take the closest power of 2, so 256. It is good to have the number of neurons in the power of 2, as it helps the computation of the network to be faster. Also, good choices for the number of neurons would be 8, 16, 32, 64, 128, 256, 512, 1024, and so on. Based on the number of input dimensions, take the number closest to 2 times the size. So, when you have 300 input dimensions, try using 512 neurons.

Number of Layers

It is true that just adding a few more layers will generally increase the performance, at least marginally. But the problem is that with an increased number of layers, the training time and computation increase significantly. Moreover, you would need a higher number of epochs to see promising results. Not using deeper networks is not an always an option; in cases when you have to, try using a few best practices.

In case you are using a very large network, say more than 20 layers, try using a tapering size architecture (i.e., gradually reduce the number of neurons in each layer as the depth increases). So, if you are using an architecture of 30 layers with 512 neurons in each layer, try reducing the number of neurons in the layers slowly. An improved architecture would be with the first 8 layers having 512 neurons, the next 8 with 256, the next 8 with 128, and so on. For the last hidden layer (not the output layer), try keeping the number of neurons to at least around 30–40% of the input size.

Alternatively, if you are using wider networks (i.e., not reducing the number of neurons in the lower layers), always use L2 regularization or dropout layers with a drop rate of around 30%. The chances of overfitting are highly reduced.



Number of Epochs

Sometimes, just increasing the number of epochs for model training delivers better results, although this comes at the cost of increased computation and training time.

Weight Initialization

Initializing the weights for your network also has a tremendous impact on the overall performance. A good weight initialization technique not only speeds up the training process but also circumvents deadlocks in the model training process. By default, the Keras framework uses glorot uniform initialization, also called Xavier uniform initialization, but this

can be changed as per your needs. We can initialize the weights for a layer using the kernel initializer parameter as well as bias using a bias initializer.

Other popular options to select are 'He Normal' and 'He Uniform' initialization and 'lecun normal' and 'lecun uniform' initialization. There are quite a few other options available in Keras too, but the aforementioned choices are the most popular.

The following code snippet showcases an example of initializing weights in a layer of a DNN with `random_uniform`.

```
from keras import Sequential
from keras.layers import Dense
model = Sequential()
model.add(Dense(64,activation="relu", input_dim = 32, kernel_
initializer = "random_uniform",bias_initializer = "zeros"))
model.add(Dense(1,activation="sigmoid"))
```

Batch Size

Using a moderate batch size always helps achieve a smoother learning process for the model. A batch size of 32 or 64, irrespective of the dataset size and the number of samples, will deliver a smooth learning curve in most cases. Even in scenarios where your hardware environment has large RAM memory to accommodate a bigger batch size, I would still recommend staying with a batch size of 32 or 64.

Learning Rate

Learning rate is defined in the context of the optimization algorithm. It defines the length of each step or, in simple terms, how large the updates to the weights in each iteration can be made. Throughout this book, we have ignored setting or changing the learning rate, as we have used the default values for the respective optimization algorithms, in our case Adam. The default value is 0.001, and this is a great choice for most

scenarios. However, in some special cases, you might cross paths with a use case where it might be better to go with a lower learning rate or maybe slightly higher.

Activation Function

We have a generous choice of activation functions for the neurons. In most cases, ReLU works perfectly. You could almost always go ahead with ReLU as an activation for any use case and get favorable results. In cases where ReLU might not be delivering great results, experimenting with PReLU is a great option.

Optimization

Similar to activation functions, we also have a fairly generous number of choices available for the optimization algorithm of the network. While the most recommended is Adam, in scenarios where Adam might not be delivering the best results for your architecture, you could explore Adamax as well as Nadam optimizers. Adamax has mostly been a better choice for architectures that have sparsely updated parameters like word embeddings, which are mostly used in natural language processing techniques. We have not covered these advanced topics in the book, but it is good to keep these points in mind while exploring various architectures.

Approaches for Hyperparameter Tuning

So far, we have discussed various hyperparameters that are available for our DL models and also studied the most recommended options for generic situations. However, selecting the most appropriate value for a hyperparameter based on the data and the type of problem is more of an art. The art is also arduous and painfully slow. The process of hyperparameter tuning in DL is almost always slow and resource intensive. However, based on the style of selecting a value for hyperparameter and

further tuning model performance, we can roughly divide the different types of approaches into four broad categories:

- Manual Search
- Grid Search
- Random Search
- Bayesian Optimization

Out of the four aforementioned approaches, we will have a brief look into the first three. Bayesian optimization is altogether a long and difficult topic that is beyond the scope for our book. Let's have a brief look at the first three approaches.

Manual Search

Manual search, as the name implies, is a completely manual way of selecting the best candidate value for the desired hyperparameters in a DL model. This approach requires phenomenal experience in training networks to get the right set of candidate values for all desired hyperparameters using the least number of experiments. Often this approach is highly efficient, provided you have sound experience in using them. The best approach to start with manual search is simply to leverage all the recommended values for a given hyperparameter and then to start training the network. The results may not be the best, but would definitely not be the worst. It's a good starting point for any newbie in the field to experiment with a few lowest-risk hyperparameter candidates.

Grid Search

In the grid search approach, you literally experiment with all possible combinations for a defined set of values of a hyperparameter. The name "grid" is actually derived from the gridlike combinations for the provided values of each hyperparameter. The following is a sample view of how a logical grid would look for three hyperparameters with three distinct values in each.

		Learning Rate			Optimizer			batch_size		
		0.1	0.01	0.001	Adam	Nadam	Adamax	32	64	128
Learning Rate	0.1				x	x	x	x	x	x
	0.01				x	x	x	x	x	x
	0.001				x	x	x	x	x	x
Optimizer	Adam	x	x	x				x	x	x
	Nadam	x	x	x				x	x	x
	Adamax	x	x	x				x	x	x
batch_size	32	x	x	x	x	x	x			
	64	x	x	x	x	x	x			
	128	x	x	x	x	x	x			

The approach is to try to develop a model for each of the combinations as shown in the preceding. The "x" indicates a model that will be developed with that particular hyperparameter value. For example, for learning rate (0.1), the vertical column shows the different models that will be developed with different values for optimizer and the batch size. Similarly, if you take a look at the horizontal row for the hyperparameter "batch-size" = 32, the "x" in all cells in the row indicates the different models that will be developed with different learning rate and optimizer values. So, in a grid with just three hyperparameters and three values each, we are looking at developing too many models. This process will be painfully long if we are developing fairly large networks and using larger training data samples.

The advantage of this approach is that it gives the best model for the defined grid of hyperparameters. However, the downside is that if your grid doesn't have great selections, your model will also not be the best one. It is simply assumed that the scientist working on the model has a fair idea of which ones could possibly be the best candidates for a given hyperparameter.

Keras doesn't directly provide the means to perform grid search tuning on the models. We can however use a custom for loop with the defined values for training or alternatively use the sklearn wrapper provided by Keras to package the model in an sklearn type object and then leverage

the grid search method in sklearn to accomplish the results. The following code snippet showcases the means to use grid search from the sklearn package by using the Keras wrapper for a dummy model.

```
from keras import Sequential
from sklearn.model_selection import GridSearchCV
from keras.wrappers.scikit_learn import KerasClassifier
from keras.layers import Dense
import numpy as np

#Generate dummy data for 3 features and 1000 samples
x_train = np.random.random((1000, 3))

#Generate dummy results for 1000 samples: 1 or 0
y_train = np.random.randint(2, size=(1000, 1))

#Create a python function that returns a compiled DNN model
def create_dnn_model():
    model = Sequential()
    model.add(Dense(12, input_dim=3, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam',
    metrics=['accuracy'])
    return model

#Use Keras wrapper to package the model as an sklearn object
model = KerasClassifier(build_fn=create_dnn_model)

# define the grid search parameters
batch_size = [32,64,128]
epochs = [15, 30, 60]

#Create a list with the parameters
param_grid =  {"batch_size":batch_size, "epochs":epochs}
#Invoke the grid search method with the list of hyperparameters
```

```
grid_model = GridSearchCV(estimator=model, param_grid=param_
grid, n_jobs=-1)
#Train the model
grid_model.fit(x_train, y_train)

#Extract the best model grid search
best_model = grid_model.best_estimator_
```

Random Search

An improved alternative to grid search is random search. In a random search, rather than selecting a value for the hyperparameter from a defined list of numbers, like learning rate, we can instead choose randomly from a distribution. This is, however, only possible for numeric hyperparameters. So, instead of trying a learning rate of 0.1, 0.01, or 0.001, it can alternatively pick up a random value for learning rate from a distribution we define with some properties. The parameter now has a larger range of values to experiment with and also much higher chances of getting better performance. It overcomes the disadvantage of a human guessing the best value for the hyperparameter confined within the defined range by inducing randomness to bring the chance for better hyperparameter selection. In reality, for most practical cases, random search mostly outperforms grid search.

Further Reading

To explore some more concrete examples and a brief guide toward Bayesian Optimization, please refer the following:

- https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f

- https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/

Model Deployment

Now, we can finally discuss a few important pointers on model deployment. We started with learning Keras and DL, experimented with actual DNNs for regression and classification, and then discussed tuning hyperparameters for improved model performance. We can now discuss a few guidelines for deploying a DL model in a production environment. I want to clarify that we won't actually be learning the process of deploying a model in production as a software engineer or discuss the DL software pipeline and architecture for a large enterprise project. We will instead focus on a couple of important aspects to be kept in mind while deploying the actual model.

Tailoring the Test Data

Throughout the course of this book, we have seen the test data exactly in sync with the train data. In this book and for that matter in any ML/DL learning guide, the experiments will always have the test data ready before model training begins. We generally split the existing data into train and test samples and then use the test data at our end to validate the model's real performance. This is a fair process as long as your objective ends with training and developing a model. Once your trained model goes live in a software, you don't really have access to the test data. To actually make use of the model, the data needs to be tailored in the expected format so that the model can predict and return the predictions. This process is actually arduous and requires carefully designing the data wrangling and transformation pipeline for production software.

Let's understand this process with an example. Assume that you have designed and developed a DNN to predict a credit card transaction as "genuine" or "fraudulent" using a supervised classification model. While developing the model, you have access to customer data, transactions, point-of-sale attributes, time-related attributes, geographical attributes, and so on. All these data points exist in different sources. For development of the model, you would make the effort to get the data from these different sources and bring it in a unified form. For your experiment, this would actually be a one-time effort. In reality, once the model is live, this entire process needs to be designed in a way that it can replicate the ingestion of data for a given customer along with all other necessary attributes from different sources, unify and transform it into the required form for the model to predict, and then make inferences at scale. Think about a large bank, where the real-time application is catering to thousands of transactions at the same time across the globe. Getting the data tailored for inferencing from the model requires really sound engineering principles to enable the model to work without glitches.

The design principles of setting up the database or a cluster/node that will compute the query request in real time need to consider the data engineering and transformations that you have done on your training dataset, because that exact same process needs to be executed every time a prediction is supposed to be made using the model. This process of tailoring the data on the fly to make inferences is a totally different art on its own and requires careful engineering to build up. Usually, data scientists are least worried about this part of the puzzle. We dispose of it under the assumption that it is a software and data engineer's job and that we can just stop bothering with it. This myth will eventually be exploded, as there is a serious harmony that needs to be established to get this part of the puzzle in place. The two teams, namely, data scientists and software engineers, need to work hand in hand to accomplish this task. The difficulty faced by a data scientist in understanding a software engineer's requirements and vice versa led to the rise of a new role in the industry called ML engineer. An ML engineer is a candidate who has a great understanding of the intersection of the two fields.

Saving Models to Memory

Another useful point we didn't discuss during the course of this chapter is saving the model as a file into memory and reusing it at some other point in time. The reason this becomes extremely important in DL is the time consumed in training large models. You shouldn't be surprised when you encounter DL engineers who have been training models for weeks at a stretch on a supercomputer. Modern DL models that encompass image, audio, and unstructured text data consume a significant amount of time for training. A handy practice in such scenarios would be to have the ability to pause and resume training for a DL model and also save the intermediate results so that the training performed up to a certain point of time doesn't go to waste. This can be achieved with a simple callback (a procedure in Keras that can be applied to the model at different stages of training) that would save the weights of the model to a file along with the model structure after a defined milestone. This saved model can later be imported again whenever you want to resume the training. The process continues just like you would want it to. All we need to do is take care of saving the model structure as well as the weights after an epoch or when we have the best model in place. Keras provides the ability to save models after every epoch or save the best model during training for multiple epochs.

An example of saving the best weights of a model during training for a large number of epochs is shown in the following snippet.

```
from keras.callbacks import ModelCheckpoint

filepath = "ModelWeights-{epoch:.2f}-{val_acc:.2f}.hdf5"
checkpoint = ModelCheckpoint(filepath, save_best_only=True,
monitor="val_acc")

model.fit(x_train, y_train, callbacks=[checkpoint],epochs=100,
batch_size=64)
```

As you can see in this code snippet, we define a `callbacks` object with the desired parameters. We define when to save the model and what metric to measure and where to save the model. The file path uses a naming convention where it stores the model weights into a file with the file name depicting the epoch number and the corresponding accuracy metric. Finally, the `callbacks` object is passed into the model fit method as a list.

Alternatively, you can also save a model in its entire form after finishing training using the `save_model` method and later load it into memory (maybe the next day) using the `load_model` method. An example is shown in the following code snippet.

```
from keras.models import load_model
#Train a model for defined number of epochs
model.fit(x_train, y_train, epochs=100, batch_size=64)

# Saves the entire model into a file named as  'dnn_model.h5'
model.save('dnn_model.h5')

# Later, (maybe another day), you can load the trained model
for prediction.
model = load_model('dnn_model.h5')
```

Retraining the Models with New Data

When you deploy your model into production, the ecosystem will continue to generate more data, which can be used for training your models again. Say, for the credit card fraud use case, you trained your model with 100K samples and got a performance of 93% accuracy. You feel the performance is good enough to get started, so you deploy your model into production. Over a period of one month, an additional 10K samples are available from the new transactions made by customers. Now you would want your model to leverage this newly available data and improve its performance even

further. To achieve this, you don't need to retrain the entire model again; you could instead use the pause-and-resume approach. All you need to do is use the weights of the model already trained and provide additional data with a few epochs to pass and iterate over the new samples. The weights it has already learned don't need to be disposed; you can simply use the pause-and-resume formula and continue with the incremental data.

Online Models

An immediate question you may ponder after understanding the process of retraining the model is how frequently should you do this: is it a good approach to retrain every day, every week, or every month? The right answer is to retrain as frequently as you want. There is no harm in incrementally training your models every time a new data point is available as long as the computation required is not a bottleneck. A good practice would be to iterate a training instance as soon as a new batch of samples is available. So, if you have set a `batch_size` of 64, you could automate the model training to ingest the newly available batch of data and further improve performance on future predictions by automating the software infrastructure to train the model for every new batch of data samples. An extremely aggressive way to keep the model performance at the best would be to incrementally train with every new data point and add previous samples as the remainder of the batch. This approach is extremely computation intensive and also less rewarding. This approach of becoming ultra-real-time and incrementally training for every new sample instead of a batch is usually not recommended.

Such models, which are always learning as and when a new batch of data is available, are called online models. The most popular examples of online models can be seen on your phone. Features like predictive text and autocorrect improve dramatically over time. If you generally type in a specific style, say combining two languages or shortening few words or using slang and so on, you will notice that the mobile phone quite actively

tends to adapt to your style. This happens purely because the phone's operating system in the background initiates the mechanism for online models to learn constantly and improve.

Delivering Your Model As an API

The best practice today in delivering your model as a service to a larger software stack is by delivering it as an API. This is extremely useful and effective, as it completely gets rid of the tech-stack requirements. Your model can easily collaborate between a diverse and complex set of components in a software ecosystem where you can worry less about the language or framework you used to develop the model. Often, when you develop an ML or DL model, the choices to deliver the model are solely driven by two simple points:

- Build the model in a language that the software engineer understands

or

- Use an API

While Python and Keras are almost universal in today's modern tech stack, we can still expect a few exceptions where this choice might not be an easy option to integrate. Therefore, we can always choose API as the preferred mode of deployment for a DL model and define the requirements for data and calling style of the API appropriately.

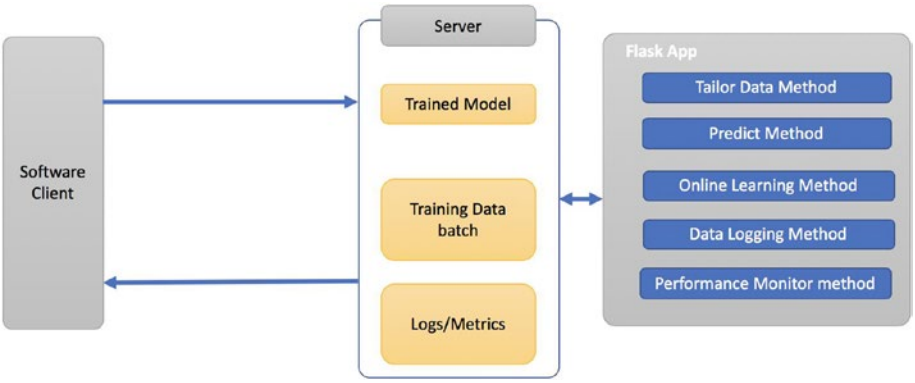
There are two extremely useful and easy-to-operate options for deploying your service as an API. You could either use Flask (a lightweight Python web framework) or Amazon Sagemaker (available on AWS). There are other options too, and I encourage you to explore them. There is an extremely well-written article on Keras Blogs on deploying your DL model using Flask.

You can explore more on this here: https://blog.keras.io/building-a-simple-keras-deep-learning-rest-api.html.

Also, you can explore how to deploy your model as an API using AWS Sagemaker in a few steps here: https://docs.aws.amazon.com/sagemaker/latest/dg/how-it-works-hosting.html.

Putting All the Pieces of the Puzzle Together

Well, to conclude, we can gather all these small components we learned in the last section and put them together into a simple (small) architecture as shown in the following figure.



This is definitely an oversimplified explanation for producing your model; I recommend you to explore improved architectures for your use case in a more appropriate way. A ton of things will change the moment you have the scale, data volume, security, and availability at increased levels. The preceding visual showcases an architecture that works for small software. Once you are done with the model-building process, you can set up most of your logic to predict, tailor the data, measure performance periodically, automate online learning, logging, and so on into a small Flask app and run it on a server and deploy it as an API. The software client, which could be a web client or another service running on the same

server, could leverage the model just by calling the API in the defined format. This architecture is okay for just small Proof of Concept (POC) and not recommended for production enterprise applications. Discussing large-scale deployment of DL models, the art of tailoring data on the fly, enabling online learning, and scaling the entire service would basically require a few more books.

Summary

In this chapter, we discussed the methods and strategies to look forward to when the model performance doesn't align with your expectations. In a nutshell, we studied the methods to incorporate when your DL model is not working well. We discussed regularization and hyperparameter tuning and also explored different strategies you could use to tune the hyperparameters and get an improved model. Lastly, we discussed a few principles we would need to address while deploying the model. We looked into an overview of the data-tailoring process for model prediction, understood how models can be trained using a pause-and-resume approach, and studied online models and the approaches to retrain them. Finally, we also looked at the options we can use to deploy the model and looked into a baby architecture for deploying the model using Flask.