DEEP LEARNING WITH KERAS

Beginner's Guide To Deep Learning With Keras

FRANK MILLSTEIN

Deep Learning with Keras

Beginner's Guide To

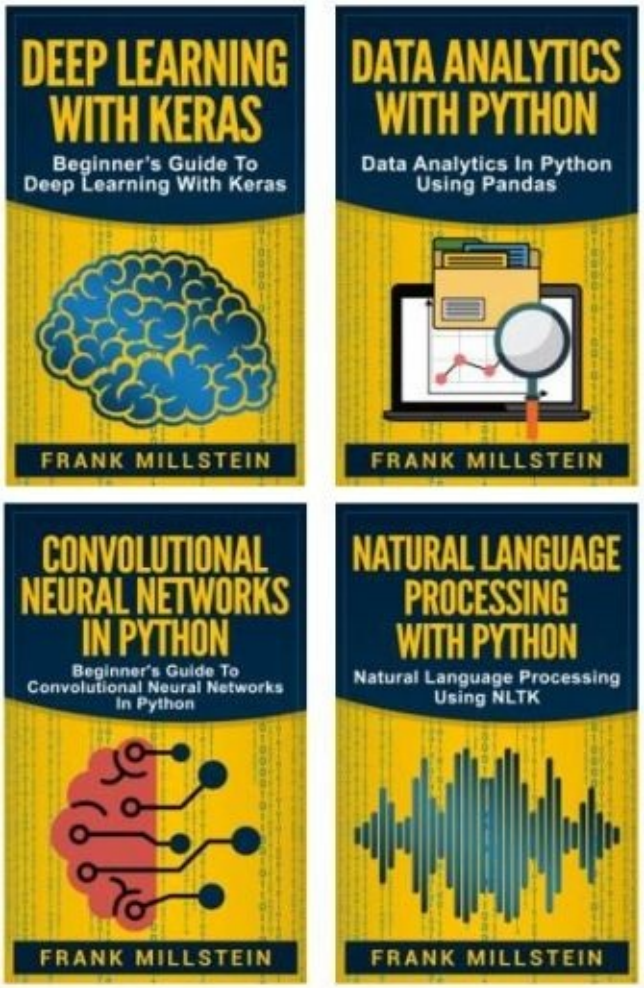
Deep Learning With Keras

By Frank Millstein

Please Check Out My Other Books Before You Continue

Below you will find just a few of my other books that are popular on Amazon and Kindle. Simply click on the link or image below to check them out.

Link To My Author Page



WHAT IS IN THE BOOK?

INTRODUCTION

Neural networks and deep learning are increasingly important studies and concepts in computer science with amazing strides being made by major tech companies like Google. Over the years, you may have heard words like backpropagation, neural networks, and deep learning tossed around a lot. Therefore, as we hear them more often, there is little wonder why these terms have seized your curiosity.

Deep learning is an important area of active research today in the field of computer science. If you are involved in this scientific area, I am sure you have come across these terms at least once. Deep learning and neural networks may be an intimidating concept, but since it is increasingly popular these days, this topic is most definitely worth your attention.

Google and other large global tech companies are making great strides with deep-learning projects, like the Google Brain project and its recent acquisition called DeepMind. Moreover, many deep learning methods are beating those traditional machine learning methods on every single matric.

HOW DEEP LEARNING IS DIFFERENT FROM MACHINE LEARNING

Before going further into this subject, we must take a step back, so you get to learn more about the broader field of machine learning. Very often, we encounter problems for which it is hard to write a computer program for solving those issues. For instance, if you want to program your computer to recognize specific handwritten digits that you may encounter on certain issues, you can try to devise a collection of rules to distinguish every individual digit. In this case, zeros are one closed loop, but what if you did not perfectly close this loop. On the other hand, what if the right top of your loop closes on that part where the left top of your loop starts?

Issues like this happen routinely, as zero may be very difficult when it comes to distinguishing from six algorithmically. Therefore, you have issues when differentiating zeroes from sixes. You could establish a kind of cutoff, but you will have problems deciding the origination of the cutoff in the first place. Therefore, quickly it becomes very complicated to compile a list of guesses and rules that will accurately classify your handwritten digits.

There are many more kinds of issues that fall into this category such as comprehending speech, recognizing objects, and understanding concepts. Therefore, we can have issues when writing computer programs, as we do not know how this is done by human brains. Despite the fact you have a relatively good idea on how to do this, your program may be very complicated.

Therefore, instead of writing a program, you can try and develop an algorithm which your computer can use to look at thousands of examples and correct answers. Therefore, your computer can use the experience that has been previously gained to solve the same problem in numerous other situations. Our main goal with this subject is to teach our computers to solve by example in the very similar way you can teach your child to distinguish a dog from a cat.

Deep learning was first theorized back in the early 1980s and was one of the main paradigms for performing broader machine learning. Over the past few decades, computer scientists have successfully developed a wide range of different algorithms which try to allow computers to learn to solve problems through examples. Because of the flurry of modern technological advancements and modern research, deep learning is on the rise since it has proven to be extremely good when it comes to teaching our computers to do what the human brain can do naturally and effortlessly.

One of the main challenges with traditional machine learning models is a process named feature extraction. More specifically, the programmers must tell the computer what kind of features and information it should be looking for when trying to make a choice or decision.

However, feeding the algorithm raw data in fact rarely works, so this process of feature extraction is one of the critical parts of the traditional machine learning workflow. Moreover, this places a massive burden on the programmer as the effectiveness of the algorithm mainly relies on how the insight of the programmer. For more complex issues, such as handwriting recognition or object recognition, this is one of the main challenges.

Fortunately, we have deep learning methods by which we can surely circumvent these challenges regarding feature extraction. This is mainly because deep learning algorithms are capable of learning to focus only on those informative, right features by themselves while at the same time they require very little guidance from the programmer. Moreover, this makes deep learning an amazingly powerful tool for machine learning.

Machine learning uses our computers to run predictive models, which are capable of learning from already existing data to forecast future outcomes, behaviors, and trends. On the other hand, deep learning is an important subfield of machine learning in which algorithms or models are inspired by how the human brain works. These deep learning models are expressed mathematically where parameters that define mathematical models can be in the order of several thousand to millions. In deep learning models, everything is learned automatically.

Moreover, deep learning is one of the main keys enabling artificial intelligence powered technologies that are being developed around the globe every day. In the following sections of the book, you are going to learn how to build complex models which help machines solve distinct real-world issues with human-like intelligence. You will learn how to build and derive many insights from these models using Keras running on your Linux machine.

The book, in fact, provides the level of details needed for data scientists and engineers to develop a greater intuitive understanding of the main concepts of deep learning. You will also learn powerful motifs that can be used in building numerous deep learning models and much more.

Machine learning and deep learning have one thing in common, that is they are both related to artificial intelligence. Artificial intelligence regards computer systems, which mimic or replicates human intelligence, while broader field of machine learning allows machine to learn entirely on their own. On the other hand, deep learning regards many computer algorithms, which attempt to model high-level abstractions contained in data to determine high-level meaning.

For instance, if artificial intelligence is used to recognize emotions in pictures, then machine learning models would input hundreds or thousands of pictures of human faces into the system while deep learning will help that system to recognize countless patterns in the human faces and the emotions they share.

This is a very simple explanation of the three. However, it is more complex. Deep learning by far is the most confusing as it works with neural networks, data, and math. Unlike deep learning, machine learning analyzes, crunches numbers and data, learns from it and uses that info to make innumerable predictions, truth statements and determinations depending on the scenario.

In this case, the machine is being trained or it is training itself on how to perform tasks correctly after learning from numbers and data it has previously analyzed. Therefore, machine learning models, builds their own solutions and logic. Machine learning can be done with several algorithms like random forest and decision tree used by Netflix for instance, that suggest movies to its customers based on their star ratings.

Another common machine learning model is a linear regression that predicts the

value of a multitude of categorical outcomes with limitless results like figuring out how much money you can sell your car for based on the current market flow. Other machine learning models include logistic regression that predicts the value of categorical outcomes based on a limited number of possible values.

Classification and naive Bayes additionally include machine learning models. Machine learning classification puts data into distinct groups like emails or filing documents while naïve Bayes includes a family of algorithms, which all share common principles in which every feature is being classified independently of other features. This may go on as there are many other machine learning models. There are two types of machine learning models as well including supervised and unsupervised.

Supervised learning models require a human to input both the data and the solution. On the other hand, these models allow the machine to figure out on its own the relationship between the two. On the other hand, unsupervised machine learning models include putting in random data and numbers for a specific situation and asking the machine or computer to find a relationship and solution. Therefore, machine learning eliminated the need for someone to constantly analyze or code data to present logic and a solution.

DEEPER INTO DEEP LEARNING

Unlike machine learning, deep learning crunches more data, which is the biggest difference between the two. For instance, if you have a little bit of data to analyze, the way to go is machine learning. However, if you have more data to analyze, deep learning is your solution. Deep learning models are extremely powerful, and they need a lot of data to give you the best possible outcome or solution. On the other hand, deep learning models need more powerful machines while machine learning models do not.

More powerful machines are required for deep learning as deep learning models do more complicated things such as matrix multiplications that require a GPU or graphics processing unit. Deep learning models also try to learn high-level features, so in the case of facial recognition, the deep learning model will get the image that is quite close to the RAW version, while a machine learning model will get a blurry image. Other powerful deep learning features are forming end-to-end solutions instead of breaking issues and solutions down in parts.

Deep learning is one of the most powerful tools used by major global tech companies. Deep learning takes a long time in processing data and finding correct solutions. Just keep in mind, it may be challenging at the very beginning, but you will get there eventually. Fortunately, you have the book to start off your deep learning journey.

CHAPTER 1: A FIRST LOOK AT NEURAL NETWORKS

In recent years, neural networks, or more specifically deep neural networks, have won numerous contests in machine learning and pattern recognition. Deep learners are mainly distinguished by the depth of their paths that are chains of the possibly learnable causal relationships between effects and actions.

Deep learning algorithms in very simple words are deep, large artificial neural nets. An NN or neural network can be easily presented as a directed acrylic graph in which the initial input layer by itself takes in signal vectors in addition to including single or multiple hidden layers and then process the outputs of those previous layers.

In fact, the main concept behind neural networks can be traced to half a century ago. There is more talk about the idea today because we have a lot more data and we have significantly more powerful computers, which were not available decades ago.

A deep neural network has many more layers and many more nodes in every layer which result in exponentially more parameters to tune. In the case when we do not have enough data, we are not able to learn those important parameters efficiently. In addition, without powerful machines or computers, learning would be insufficient as well as too slow.

When it comes to small datasets, traditional machine learning algorithms such as random forests, regression, GBM, SVM and statistical learning do an amazing job. However, when the data scale goes up containing a large amount of information, those large, deep neural networks quickly outperform the traditional ones.

This happens primarily because compared to a traditional machine learning algorithm, a deep neural network model has a wider range of parameters and it has the capability of learning more complex nonlinear parameters. Therefore, we expect a deep neural network model to automatically pick the most important and helpful features on its own without too much manual engineering.

As already mentioned, deep learning is one of the main forms of machine learning which uses a model or algorithm of computing that is very much based or inspired by the structure of the human brain. Therefore, these models are called neural networks. The basic unit of any neural network is the neuron. Every neuron has a specific set of inputs. In addition, every neuron has a specific weight. The neurons have the power of computing functions based on these specific weighted inputs. For instance, a linear neuron takes a linear combination of its weighted inputs while sigmoidal neurons feed the specific weighted sum of its inputs into a logistic function.

The logistic function always returns a value between 1 and 0. In the case when the weighted sum is negative, the return value is close to 0. On the other hand, when the weighted sum is large or positive, the return value is close to 1. When it comes to mathematical problems, the logistic function is a logical choice since it has nice looking derivatives that make the learning process simpler.

However, whatever function the neuron uses, the value computed is immediately transmitted to other included neurons that show the neuron's output. In practice, these sigmoidal neurons are used more often than linear functions since they enable more versatile deep learning models in comparison to linear neurons.

A deep neural network occurs when you start connecting neurons to each other in your input data and eventually to the outlets that correspond to your network's answer to your learning problems. To make this structure easier to visualize, you must obtain the weight of your neurons in the link that connect your initial layer to other layers contained in your neuron.

Very similar to how neurons are organized in layers in the brain, neurons in deep neural nets are typically organized in certain layers. In deep neural networks, neurons situated on the bottom layers are those that receive signals from the inputs. While neurons situated in the top layers are connected to the answer; thanks to their outlets. Usually, there are no connections between neurons in the same layers as more complex connectivity between neurons require more mathematical analysis.

In the case when there are no connections, which lead from a neuron in a higher level to those neurons in lower layers, we call them feed-forward neural networks. Opposed to these neural networks are recursive neural networks that are much more complicated to train and analyze. Now, we will go through several of the most commonly used deep neural networks.

CONVOLUTIONAL NEURAL NETWORK

Convolutional neural networks or also known as CNN and are one of the most commonly used types of feed-forward neural networks in that the connectivity pattern between neurons is based on the organization of the common visual cortex system. Therefore, the V1 or the primary visual cortex does edge detection out of raw visual input obtained from the retina. Then, the V2 or secondary visual cortex receives the edge features from the primary visual cortex and extracts simple visual properties such as spatial frequency, orientation, and color.

The visual area of V4 or another visual cortex mainly handles more complicated attributes or grained objects. Then, all those processed visual features flow into the final unit named inferior temporal gyrus of IT for further object recognition. This specific shortcut between V1 layer and V4 layer, in fact, inspired a certain type of convolutional neural networks with connections between those non-adjacent layers named residual net. Residual nets contain residual blocks that support inputs of one layer to be readily passed to those layers coming later.

Therefore, convolutional neural networks are commonly used for edge detections, extracting simple visual properties such as spatial frequency, orientation, and colors, detecting object features of intermediate complexity and object recognition.

Convolution is commonly used in mathematical terms referring to an operation between matrices as convolutional layers which generally have a small matrix named filter or kernel. As the filter or kernel is sliding or convolving across

these matrices of input images, it is, at the same time, computing the important element-wide multiplication of specific values contained in the kernel matrix as well as contained in the original image values.

Therefore, specifically designed filters or kernels are capable of processing images for very specific purposes such as image sharpening, blurring, edge detection and many other processes efficiently and rapidly.

RECURRENT NEURAL NETWORK

A neural network sequence model is commonly designed to transform input sequences into an output sequences, which live in a different domain. Another common type of deep neural networks named RNN or recurrent neural networks are greatly suitable for these purposes as they have shown an amazing improvement in problems like speech recognition, handwriting recognition, and machine translation.

An RNN model is born with an amazing capability of processing long sequential data and tackling very complex tasks with context spread over a period. The recurrent model, in fact, processes single element in the neural sequence at the time. After the initial computation, this newly updated unit state is easily passed down to that next time step to facilitate the computation of every next element.

Imagine the case when recurrent neural network model reads all articles on Wikipedia character by character. On the other hand, simple perceptron neurons which linearly combine the current input elements, as well as the last unit state typically, may lose those long-term dependencies.

For instance, we can start a sentence with Susan is working at… Then, after a whole paragraph, we want to start our next sentences with He or She correctly. If the recurrent neural model forgets the character's name we used, we can never know. To resolve this issue, engineers have created a special deep neuron that comes with more complicated internal structure designed to memorize the long-term context named LTSM or long-short-term memory.

LTSM models are smart enough to learn long-term context. These models can learn for how long they should memorize the old information, when to forget information, when to use newly updated data, and when to combine the new input with old memory. Using the power of LSTM and RNN cells, you can build an RNN character-based model that will be able to learn the specific relationship between the characters to form words and sentences without any previous knowledge of English vocabulary. This RNN model, in fact, could achieve a very good performance even without a large set of training data.

RNN SEQUENCE TO SEQUENCE MODEL

The common sequence-to-sequence model is very often used as an extended version of recurrent neural models, but its application field is more distinguishable. Same as recurrent neural networks, sequence-to-sequence models operate on sequential data, but they are commonly used to develop personal assistants or chatbots by generating meaningful responses to numerous input questions.

Some common sequence-to-sequence models consist of two recurrent neural networks including decoder and encoder. In this case, the encoder learns everything about the obtained contextual information from various input words. Then, the encoder hands this knowledge down to the decoder by using a specific context vector also known as thought vectors. Eventually, the decoder consumes these context vectors and generates correct responses.

AUTOENCODERS

Autoencoders are different from the previous deep learning models as they are used only for unsupervised deep learning. Autoencoders are designed mainly to learn low-dimensional representation of high-dimensional data sets very similar to what PCA or principal components analysis does. The autoencoder model is capable of learning approximation functions to reproduce the input data.

On the other hand, specific bottleneck layers situated in the middle containing a very small number of nodes restrict these models. With this very limited number of nodes, these models come with very limited capacity, so they are forced to form a specific, very efficient encoding of the data, which is the low-dimensional code we obtained.

You can use autoencoder models to compress your documents on a variety of topics. There are some limitations as these models as they come with a bottleneck layer that contains only a few neurons.

However, when you use both autoencoder and PCA, you can reduce your documents onto two dimensions, so your autoencoder model will demonstrate a much better outcome. With the help of these models, you can do very efficient data compression to speed up the overall process of information retrieval including both images and documents.

REINFORCEMENT DEEP LEARNING

Reinforcement learning is one of the secrets behind many successful AI projects done in the past. Reinforcement learning is a subfield of machine learning that allows software agents and machines to automatically determine the optimal behavior within a given context with the main goal of maximizing the long-term performances, which are measured by a given metric.

Most reinforcement learning projects start with a specific supervised learning process, train a fast rollout policy as well as policy network, mainly relying on the manually curated training data. The reinforcement learning policy network gets improved when it gained more versions of the previous policy network. Therefore, with more and more obtained data, it gets stronger and stronger without requiring any additional external training of data.

GENERATIVE ADVERSARIAL NETWORK

Another type of deep neural network commonly used is generative adversarial network or GAN. This is a type of deep generative algorithm. GAN has the power of creating new examples after going through and learning some of the real data. A common type of GAN consists of two models that are competing against each other in a zero-sum network.

The generative adversarial network mainly contains some real-world examples, generator, generated fake samples, discriminator, and fine tune training. These models can tell the fake data apart from the true data due to data distribution. GAN was initially proposed to generate meaningful images after learning from real world photos.

The GAN model proposed in the original GAN paper was composed of two independent models including the discriminator and the generator. In this case, the generators produced fake images and sent the output back to the discriminator model. Then, the discriminator worked in a manner very similar to a judge since it was fully optimized to identify the fake photos from the real ones.

In addition, the generator model, at the same time, was trying hard to cheat the discriminator as the judge was trying very hard not to be cheated by the generators. This was a very interesting zero-sum game occurring between these GAN models that motivated both models to further improve their functionalities and develop their designed skills.

After learning about these deep neural network models, you probably wonder how you can implement these models and use them in real problem solving with deep learning issues. Fortunately, there are many source libraries and toolkits you can use for building your own deep learning models. TensorFlow arguably is one of the most popular that attracted a lot of attention. In terms of popularity, Theano follows TensorFlow very closely. Therefore, those two are the best numerical platforms in Python which provide the basis for innumerable deep learning projects.

Both are very powerful libraries and can be used for different tasks for creating deep learning models. Another powerful tool in Python's library is Keras which we are going to use in this book. Keras is an amazingly, powerful high-level neural network API with astonishing power of running on top of Theano, TensorFlow or CNTK. It was written in Python and developed with the focus of enabling fast and efficient deep learning experimentation.

CHAPTER 2: GETTING STARTED WITH KERAS

We are going to use Keras as it allows fast and easy prototyping, supports both recurrent and convolutional networks and a combination of the two, and runs seamlessly on GPU and CPU. Designed to enable fast deep learning modeling and experimentation with neural networks, it focuses on being modular, minimal and extensible.

Therefore, with Keras you can build a wide range of different deep learning models, which run on top of TensorFlow or Theano effortlessly and efficiently. Keras is a free open-source neural network, so you will find and install it easily. The core data structure of Keras is a model that is a way of organizing multiple layers. Before you delve deeper into Keras, you must install it, of course. Be aware that this popular programming deep learning framework uses either Theano or TensorFlow behind the scenes instead of providing all the functionality by itself.

Keras is very simple to install if you have been working in SciPy and Python environment. Make sure you have an installation of TensorFlow or Theano on your system already before you install Keras. Keras can be very easily installed using PyPI.

sudo pip install keras

python -c "import keras; print keras. __ version __ "

1.1.0

sudo pip install - - upgrade keras

Using the same method, you can upgrade your version of Keras. Assuming you have installed both TensorFlow and Theano, you are able to configure the backend used by Keras. The best way is by editing or adding the Keras configuration file in your directory.

~/. keras / keras . json

{

" image dim ordering ": " tf ",

" epsilon " : 1e - 07,

" floatx " : " float32 ",

" backend " : " tensorflow "

}

In this configuration file, you can change the property of backend from TensorFlow to Theano. In this case, Keras will use your new configuration the next time you run it. In addition, you can easily configure Keras as follows.

python - c " from keras import backend ; print backend. _ BACKEND "

Using TensorFlow backend.

Tensorflow

In addition, you can specify which backend you want Keras to use on your command line as shown below.

KERAS _ BACKEND = theano python - c " from keras import backend ; print(backend. _ BACKEND) "

Running this script, you get as illustrated below.

Using Theano backend.

theano

BUILDING DEEP LEARNING MODELS WITH KERAS

The focus of Keras is a model. The main kind of model build in Keras is called a sequence containing a linear stack of multiple layers. Therefore, you create a sequence and gradually add layers to the model in the order you want so you get proper computation. Once you define your model, you must compile your model so that it makes use of the specific underlying framework to optimize the entire process of computation, which will be performed on your deep learning model.

In this case, you must specify the optimizer and the loss function, which will be used. Once you compile your model, your model must fit to the data. This is done one by one, one batch of data at a time. In fact, this is where all the computation occurs. Once you train your model, you can use it to make other new predictions on your data.

In summary, the construction of deep learning models in Keras can be explained as defining your model, compiling your model, fitting your model and making predictions. To define it, you must create a sequence and add multiple layers. Once done, you compile your model by specifying optimizer and loss functions. Then, you must fit your model by executing the model using data. Finally, you make predictions on new data.

As already mentioned, Keras is amazingly powerful and easy to use for evaluating and developing varied deep learning models. It wraps all those efficient numerical computation libraries like TensorFlow and Theano and allows you to define and properly train your neural network models in several

lines of code.

Following, you are going to learn how to create your first network model in Python using Keras. Before you begin, make sure you have Python 2, or a newer version installed and configured. You also need NumPy and SciPy installed and configured, and, of course, you need to have Keras and TensorFlow or Theano installed and configured. Once you have these up and running, create a new file as follows.

keras _ first _ network . py

In the following section, you will learn how to load data, define your model, compile model, fit model, evaluate your model, and tie it all together to work perfectly on your future models.

Whenever we work with deep learning models which use a stochastic process like random numbers, it is a very good idea to set the random number seed. This is a good idea as you will be able to run the same code over and over and get the same result. This is also very useful in the case when you need to demonstrate a result, compare different models using the same source, or when you need to debug a part of your code. For the initialization of the random number generator, use the following script.

```
from keras . models import Sequential
from keras . layers import Dense
import numpy
```

numpy . random . seed (7)

Once done, you can load your data. To do so in this example, we are going to use a very popular Pima Indians onset which is a standard deep learning dataset developed by the UCI Machine Learning repository. It describes patient medical data for Pima Indians within five years. Therefore, this is a binary classification problem as all the input variables used are numerical. Therefore, this really makes it easy to use this dataset directly with a neural network in Keras. To use it, download the dataset and place it in your working directory, which is the same as your previously created Python file.

Now, we continue with building your model with Keras. You must load the file directly using the specific NumPy function. There is one output variable in the last column and eight input variables. Once you load the data, you can split your dataset into output variable denoted as Y and input variables denoted as X.

dataset = numpy . loadtxt("pima – indians –diabetes . csv", delimiter = ",")

X = dataset [:,0:8]

Y = dataset [:,8]

Make sure you have initialized your random number generator to ensure your results are reproducible as well as properly loaded on your data. Now, you must define your neural network model.

As already mentioned, models in Keras are defined as a specific sequence of multiple layers. To create a sequential model and then add one layer at a time

until you are satisfied with your network topology, you must define your model. The first thing you must do is to ensure your input layer has the proper number of inputs. You can specify this when you create your first layer using the input dim argument. Make sure you set it to eight for the eight input variables.

Now you probably wonder, how do you know the right number of layers and their types. Well, this is a complex question. There are some heuristics you can use. However, the best network structure is found through a certain process of trial and error. You will need a network large enough to capture the core structure of the problem.

Further, we are going to see a fully-connected neural network structure containing three layers. Take into consideration that fully-connected layers are mainly defined using the dense class. You can specify the right number of neurons contained in the layer as your first argument, while your second argument is defined as init. Then, you can specify your activation function as well using the activation argument.

In the following case, we are going to initialize the specific network weights to your small random number generated from a specific uniform distribution. In this case, you will get between 0 and 0.05 as this is the default uniform weight when using Keras. There is also another alternative named normal and invariably used for small random numbers that are generated from Gaussian distribution.

In our example, we are going to use the relu or rectifier activation function as well on the two initial layers. We must use the sigmoid function in our output

layer. Before, it was common to use the tanh and sigmoid activation functions for all layers.

However, these days this is not the case as better performance is achieved when you use the rectifier activation function in addition to using a sigmoid function on your output layer to ensure your neural network output is between one and zero. Using this manner, it is very easy to map all classes with a default threshold. Then, you can piece everything together by adding layers. Your first layer will have twelve neurons, so expect eight input variables. Your second hidden layers will have eight neurons while your output layer will have one neuron predicting the classes.

```
model = Sequential( )
model . add ( Dense ( 12, input _ dim = 8, activation = ' relu '))
model . add ( Dense ( 8, activation = ' relu '))
model . add ( Dense ( 1, activation = ' sigmoid '))
```

The next step is to compile your model. To compile your neural network model, you must use the efficient numerical libraries called backend like TensorFlow or Theano.

When you use it for compiling, the backend automatically chooses the best possible way for representing your network for making predictions and training, which will run on your hardware like GPU or CPU and sometimes even distributed.

When compiling your model, firstly, you must specify some additional properties, which are required when you train your neural network. Be cognizant

that training your network means finding the best possible set of weights which you use for making right predictions for your specific problem.

To evaluate a set of weights, firstly, you must specify the loss functions. You will use the optimizer that is used to search through different weights for your network as well as optional metrics you would like to collect and report during model training. In this specific case, you will use logarithmic loss, which is defined as binary crossentropy for most of binary classification problems.

You must use the efficient gradient descent algorithm or Adam only because it is a very efficient default. Since this is a classification problem, you must collect and report the metric of the classification accuracy.

```
model . compile ( loss = ' binary_crossentropy ', optimizer = ' adam ', metrics = [ 'accuracy' ])
```

Once you are done with compiling your model, the next step is fitting your model. Once you define and compile your model, you must fit it, so it is ready for efficient computation. Therefore, now is the right time to execute your model with the data. You can train or fit your neural network model on your loaded data by calling the specific fit function on your model.

Consider that the model training process will run for a specific number of iterations through your dataset named epochs. Therefore, you must specify your model using an epochs argument. You can set the specific number of instances, which will be evaluated before you update weight.

This is called the batch set and site you call using the batch size argument. For this specific problem, you will run a small number of iterations, hundred and fifty. You will also use a small batch size of ten. As mentioned before, these can be chosen experimentally by your model using trial and error. During this step, the computation occurs on your GPU or CPU.

model . fit(X, Y, epochs = 150, batch _ size = 10)

Once you are done with fitting your model, you must evaluate it as the next step. Up to this point, you have trained your neural network on the entire dataset, so now you can easily evaluate the overall performance of your neural network on the same dataset. This will give you the best idea on how well you have just modeled the obtained dataset also known as train accuracy.

However, you will have no idea of how well your model may perform on the new data. You have done this mainly for simplicity, but an ideal path is to separate your data into test and train datasets for evaluation and training of your model.

You can surely evaluate your model on your training dataset using the specific evaluate function on your model and then pass that same input and output you have used to train your model.

This will result in a prediction for every input and output pair, so you collect scores in patients including average loss and any other important metrics you have just configured like accuracy.

```
scores = model . evaluate (X, Y)
print( "\n%s: %.2f%%" % ( model . metrics _ names [1], scores [1]*100 ))
```

Once you tie everything together, you get your first neural network model you have just created in Keras. Your complete code will look as follows.

```
from keras . models import Sequential
from keras . layers import Dense
import numpy
numpy . random . seed (7)
dataset = numpy . loadtxt( "pima – indians –diabetes . csv", delimiter = "," )
X = dataset [:,0:8]
Y = dataset [:,8]
model = Sequential( )
model . add ( Dense( 12, input _ dim =8, activation =' relu '))
model . add ( Dense(8, activation = 'relu' ))
model . add (Dense (1, activation = 'sigmoid' ))
model . compile ( loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
model . fit ( X, Y, epochs = 150, batch _ size = 10)
scores = model . evaluate (X, Y)
print ( "\n%s: %.2f%%" % ( model . metrics _ names [1], scores [1]*100 ))
```

Running this, you should see a message for every of the hundred and fifty epochs that print both the accuracy and loss for each, followed by the final evaluation of your trained model on your training dataset. You should get message like following.

...

Epoch 145/150

768/768 [==============================] - 0s - loss: 0.5105 - acc: 0.7396

Epoch 146/150

768/768 [==============================] - 0s - loss: 0.4900 - acc: 0.7591

Epoch 147/150

768/768 [==============================] - 0s - loss: 0.4939 - acc: 0.7565

Epoch 148/150

768/768 [==============================] - 0s - loss: 0.4766 - acc: 0.7773

Epoch 149/150

768/768 [==============================] - 0s - loss: 0.4883 - acc: 0.7591

Epoch 150/150

768/768 [==============================] - 0s - loss: 0.4827 - acc: 0.7656

32/768 [>.....] - ETA: 0s

acc: 78.26%

You can use this neural network model you have just created for making predictions as well. However, you will must adapt our example from the above just a bit to use it for making predictions. Making predictions is very easy once you call model predict argument.

In this case, you are going to use a sigmoid activation function on your input layers, so you get predictions in the range between one and zero. In addition, you can quickly convert them into binary predictions for your classification task by just rounding them.

To run predictions for every record contained in your training data, you must run

code as shown below.

```
from keras . models import Sequential
from keras . layers import Dense
import numpy
seed = 7
numpy . random . seed ( seed )
dataset = numpy . loadtxt( "pima – indians –diabetes . csv", delimiter = ",")
X = dataset [:,0:8]
Y = dataset [:,8]
model = Sequential()
model . add ( Dense ( 12, input _ dim = 8, init = '  uniform ', activation = ' relu '))
model . add ( Dense (8, init = 'uniform', activation = 'relu'))
model . add ( Dense(1, init = 'uniform', activation = 'sigmoid' ))
model . compile ( loss = 'binary_crossentropy', optimizer = ' adam ', metrics = [ 'accuracy' ])
model . fit ( X, Y, epochs = 150, batch _ size = 10,  verbose = 2)
predictions = model . predict (X)
rounded = [ round (x [0])
print(rounded)
```

Running this neural network model, you will print the predictions for every input pattern obtained. In addition, you can use these obtained predictions directly in an application, if required.

As you now know how to create your neural network in Keras, we are going to move to more complex deep learning tasks that you can efficiently execute using the powerful Keras Python library.

CHAPTER 3: MULTI-LAYER PERCEPTRON NETWORK MODELS

The powerful Keras Python library for deep learning problems mainly focuses on the creation of deep learning models as a collection, a sequence of multiple layers. In the following section of the book, you are going to learn how to use simple components to create simple multi-layer perceptron network models using Keras.

As already mentioned in the book, the simplest model you can create is defined in the sequential class that is a linear stack of multiple layers. You can create a sequential model and then define all included layers as seen below.

```
from keras . models import Sequential
model = Sequential (...)
```

However, a model useful idiom is to first create your sequential model and then add layers for proper computation as illustrated below.

```
from keras . models import Sequential
model = Sequential ()
model . add (...)
model . add (...)
model . add (...)
```

Once done, you must add model inputs. Be mindful that the first layers in your neural network model must specify the shape of your input. This, in fact, is the total number of inputs attributed as defined by the argument input dim. This argument will expect an integer. For instance, you can readily define your input in terms of eight inputs for your dense layer as follows.

Dense (16, input _ dim = 8)

MODEL LAYERS

Once done, you must model layers of your neural model. Remember that layers of different kind usually have several properties in common especially their activation functions and their weight initialization functions. For your models, you must use weight initialization arguments. This kind of initialization is used for a certain layer, which is specified in the init argument.

Some of the most commonly used weight initialization arguments include uniform, normal and zero. When it comes to the uniform weight initialization, weights are initialized to small uniformly random values between 0 and 0.5. On the other hand, normal weight initialization are weights that are initialized to a small Gaussian random value. Consider that standard deviation is zero mean of 0.5. The last type is zero when all weight is set to zero values.

Keras supports many standard neuron activation functions as well such as rectifier, sigmoid, tanh and softmax. You will ordinarily specify the type of your activation functions used by a specific layer in your activation argument that takes a string value. You can create an activation object that you can add directly to your model just after you apply the activation functions to the output of the specific layer.

There is a wide range of different core layers in Keras used for standard neural networks. Some of the most useful and routinely used core layer types include dense, dropout and merge layers. Dense is a fully-connected layer used most often on multi-layer perceptron models. Dropout core layers apply dropout to the neural network model by setting a fraction of inputs to zero to reduce very

common issue, over fitting. Merge core layers combine the inputs from several Keras models into a single Keras model.

MODEL COMPILATION

As you already know, once you have done defining your model, you must compile it. The model compilation will create the highly efficient structure that will be used by the underlying backend, TensorFlow or Theano, to efficiently execute your neural network model during the training process. You can compile your neural network model using the compile argument. It will accept three important attributes of your model including loss function, model optimizer, and metrics.

```
model . compile ( optimizer = , loss = , metrics = )
```

When it comes to the model optimizers, the optimizer is the main search technique recurrently used to update weight in your neural network model. You can create an optimizer object and pass it to your compile function using the optimizer argument. This will allow you to effortlessly configure the overall optimization process with its own arguments like learning a specific rate.

```
sgd = SGD (...)
model . compile ( optimizer = sgd )
```

You can also use the default parameters of your optimizer by just specifying the name of the specific optimizer to your optimizer argument as shown below.

```
model . compile ( optimizer = 'sgd' )
```

Some frequently used gradient descent optimizers you can use include SGD, RMSprop and Adam. The SGD is usually used stochastic gradient descent with great support for momentum. The RMSprop is often used in adaptive learning rate optimization methods while Adam, short for Adaptive Moment Estimation, other adaptive learning rates.

Once you use the right optimizer, you must move to model loss functions. The loss function is also called the objective function. It is the evaluation of the neural network models used by the optimizers to navigate the weight space.

You can quickly specify the name of your loss function, which will be further used by the compile functions. Some of the most normally used loss function arguments include MSE for mean squared error, categorical crossentropy for numerous multi-class logarithmic tasks and binary crossentropy for binary logarithmic loss.

Once you obtain your model loss function, you move to metrics. Metrics are evaluated during the process of model training. Consider that only one metric is supported at the time and that is for accuracy.

MODEL TRAINING

Your neural network model is trained on NumPy arrays. You will use the fit function as seen below.

model . fit (X, y, epochs = , batch _ size =)

Model training both specifies the number of epochs you must train and the batch size. As already mentioned in the book, epochs are the total number of times your model is exposed to the dataset used for training while the batch size is the total number of training instances shown to your model before you perform weight update.

As mentioned, for model training you are going to the fit functions which allows a basic evaluation to be performed on your model during model training. You can handily set the validation split value to hold back a certain fraction of your training dataset for further validation to be evaluated by each epoch. You may use a validation data tuple of Y and X of data to evaluate. Moreover, fitting your model returns a history object with metrics and details previously calculated for the neural network model each epoch. This is invariably used for graphing your model's overall performance.

MODEL PREDICTION

Once you are done with training your neural network model, you can use it to make predictions on your test data or on other new data. There is a wide range of different output types you can calculate from your trained neural network model. Each of these models is calculated using a different function you call on your neural network model.

For instance, you can use model evaluate function to calculate the loss values of your input data or you can use model predicts to generate your network output for your input data. You have an option to use model predict proba argument to generate class probabilities for your input data or use model predict classes function to generate different class outputs for your input data. On some classification problems, you must use the predict classes argument to make different predictions for new data instances or for test data.

Once you are happy with your model and its properties, you can finalize it. You may need a summary of your model. If so, you can readily display a summary of your neural network model by calling the routinely used summary function as follows.

model . summary ()

You have an option to retrieve your model summary using the get config argument as follows.

model . get _ config ()

Finally, you have an option to create an image of your neural network model structure as seen below.

```
from keras . utils . vis utils import plot model
plot( model, to _ file = ' model . png ')
```

Therefore, in this section of the book, you discovered the Keras API, which you can use to create innumerable deep learning and artificial neural network models. You have learned how to construct a multi-layer neural network model, how to add multiple layers including activation and weight initialization. You have thus learned how to compile your neural network model using several optimizers including metrics and loss functions. Now, you know how to fit your models including batch size and epochs as well as how to make model predictions and summarize your model.

CHAPTER 4: ACTIVATION FUNCTIONS FOR NEURAL NETWORKS

In this section of the book, we are going to give more attention to most regularly used activation functions in neural networks. In this example, we are going to use the MNIST. MNIST data is a set of approximately 70000 photos of miscellaneous handwritten digits where each photo is black and white and 28x28 in size. We are going to solve this specific problem using a fully connected neural network with several different activation functions.

Our input data will be 70000,784 while our output shape will be 70000,10. Therefore, we use a fully connected neural network model with one hidden layer. There are 784 neurons contained in the input layer, one for every pixel in the photos and there are 521 neurons contained in the hidden layer. In the output layer, there are 10 neurons for every digit. Using Keras, we can utilize several different activation functions for every layer in our neural network model. This means that in this case, we must decide which activation functions should be used in the output layer and which activation function should be used in the hidden layer. There are many different activation functions, but most often used are relu, tanh and sigmoid. Firstly, we will not use any activation functions to start building a basic sequential model.

model = Sequential ()
model . add (Dense (512, input _ shape = (784,)))
model . add (Dense(10, activation = ' softmax ')) As already mentioned, in the input there

are 784 neurons, in the hidden layer there are 512, and there are 10 neurons contained in the output layer. Before you train your model, you can look at your neural network structure and parameters using model summary argument as illustrated below.

Layers (input = = > output)
---
dense _ 1 (None, 784) = = > (None, 512)
dense _ 2 (None, 512) = = > (None, 10) Summary

Layer (type) Output Shape Param #
=====
dense_1 (Dense) (None, 512) 401920
_____
output (Dense) (None, 10) 5130
=====
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_____
None Once you are sure about the structure of your model, you must train it for five epochs.

Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 3s - loss: 0.3813 - acc: 0.8901 - val_loss: 0.2985 - val_acc: 0.9178
Epoch 2/5
60000/60000 [==============================] - 3s - loss: 0.3100 - acc: 0.9132 - val_loss: 0.2977 - val_acc: 0.9196
Epoch 3/5
60000/60000 [==============================] - 3s - loss: 0.2965 - acc: 0.9172 - val_loss: 0.2955 - val_acc: 0.9186
Epoch 4/5
60000/60000 [==============================] - 3s - loss: 0.2873 - acc: 0.9209 - val_loss: 0.2857 - val_acc: 0.9245
Epoch 5/5
60000/60000 [==============================] - 3s - loss: 0.2829 - acc: 0.9214 - val_loss: 0.2982 - val_acc: 0.9185

Test loss:, 0.299
Test accuracy: 0.918

As you can see, our results of 91.8% using MNIST is quite bad. When you plot the losses, you will see that the validation loss is far away from improving and it will not improve even after hundred epochs.

Therefore, we must try different techniques to prevent a common problem of over-fitting from occurring. We need more techniques to make our neural network model learning better and working smarter. We can achieve this by using one of the most customarily used activation functions, the sigmoid activation function.

SIGMOID ACTIVATION FUNCTION

To improve our neural network model, we will use sigmoid activation function. It will squash our input into a 0,1 interval.

```
model = Sequential ()
model . add ( Dense ( 512, activation = ' sigmoid ', input _ shape = ( 784, )))
model . add ( Dense(10, activation = ' softmax '))
```

You will see that the structure of your neural network remained the same as you just have changed the activation function of your dense layer. You can try the same for five epochs.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [ ============================== ] - 3s - loss: 0.4224 - acc: 0.8864 -
val_loss: 0.2617 - val_acc: 0.9237
Epoch 2/5
60000/60000 [ ============================== ] - 3s - loss: 0.2359 - acc: 0.9310 -
val_loss: 0.1989 - val_acc: 0.9409
Epoch 3/5
60000/60000 [ ============================== ] - 3s - loss: 0.1785 - acc: 0.9477 -
val_loss: 0.1501 - val_acc: 0.9550
Epoch 4/5
60000/60000 [ ============================== ] - 3s - loss: 0.1379 - acc: 0.9598 -
val_loss: 0.1272 - val_acc: 0.9629
Epoch 5/5
60000/60000 [ ============================== ] - 3s - loss: 0.1116 - acc: 0.9673 -
val_loss: 0.1131 - val_acc: 0.9668

Test loss: 0.113
Test accuracy: 0.967
```

This looks much better. You get a linear combination of your input with the bias and the weights even after you stacked many layers. This is very similar to a neural network without any hidden layers. You can add some more layers just to see what will occur as shown below.

```
model = Sequential( )
model . add ( Dense ( 512, input _ shape = ( 784, )))

for i in range(5):
model . add ( Dense ( 512 ))

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

When you do this, you get your neural network model looking as indicated below.

```
Dense _ 1 ( None, 784 ) = => ( None, 512 )
dense _ 2 ( None, 512 ) = => ( None, 512 )
dense _ 3 ( None, 512 ) = => ( None, 512 )
dense _ 4 ( None, 512 ) = => ( None, 512 )
dense _ 5 ( None, 512 ) = => ( None, 512 )
dense _ 6 ( None, 512 ) = => ( None, 10 )

_________________________________________________________________
Layer ( type )                 Output Shape              Param #
=================================================================
dense_1 ( Dense )             ( None, 512 )             401920
_________________________________________________________________
dense_2 ( Dense )             ( None, 512 )             262656
_________________________________________________________________
dense_3 ( Dense )             ( None, 512 )             262656
_________________________________________________________________
dense_4 ( Dense )             ( None, 512 )             262656
_________________________________________________________________
dense_5 ( Dense )             (None, 512 )              262656
_________________________________________________________________
dense_16 ( Dense )             ( None, 10 )              5130
=================================================================
Total params: 1,720,330
Trainable params: 1,720,330
Non-trainable params: 0
_________________________________________________________________
None
```

You get results for five epochs as follows.

```
Train on 60000 samples, validate on 10000 samples
```

Epoch 1/5
60000/60000 [==============================] - 17s - loss: 1.3217 - acc: 0.7310 - val_loss: 0.7553 - val_acc: 0.7928
Epoch 2/5
60000/60000 [==============================] - 16s - loss: 0.5304 - acc: 0.8425 - val_loss: 0.4121 - val_acc: 0.8787
Epoch 3/5
60000/60000 [==============================] - 15s - loss: 0.4325 - acc: 0.8724 - val_loss: 0.3683 - val_acc: 0.9005
Epoch 4/5
60000/60000 [==============================] - 16s - loss: 0.3936 - acc: 0.8852 - val_loss: 0.3638 - val_acc: 0.8953
Epoch 5/5
60000/60000 [==============================] - 16s - loss: 0.3712 - acc: 0.8945 - val_loss: 0.4163 - val_acc: 0.8767

Test loss: 0.416
Test accuracy: 0.877

This is quite bad. You can see that your neural network model is just unable to learn what you want. This happened because without nonlinearity, your neural network is just a basic linear classifier unable of acquiring any nonlinear relationships.

On the other hand, sigmoid is always a nonlinear function, so we cannot represent it as a lineal combination of our input. That is exactly what brings nonlinearity to your neural network model, so it can learn any nonlinear relationships. Now, train your neural network model. Train the five-hidden layer model using sigmoid activations.

Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 16s - loss: 0.8012 - acc: 0.7228 - val_loss: 0.3798 - val_acc: 0.8949
Epoch 2/5
60000/60000 [==============================] - 15s - loss: 0.3078 - acc: 0.9131 - val_loss: 0.2642 - val_acc: 0.9264

Epoch 3/5
60000/60000 [==============================] - 15s - loss: 0.2031 - acc: 0.9419 - val_loss: 0.2095 - val_acc: 0.9408
Epoch 4/5
60000/60000 [==============================] - 15s - loss: 0.1545 - acc: 0.9544 - val_loss: 0.2434 - val_acc: 0.9282
Epoch 5/5
60000/60000 [==============================] - 15s - loss: 0.1236 - acc: 0.9633 - val_loss: 0.1504 - val_acc: 0.9548

Test loss: 0.15
Test accuracy: 0.955

This is much better. In this case, you are probably over-fitting, but you can see that you got a great boost in your model's performance just by using the activation function. Sigmoid activation functions are great as they have many phenomenal properties like differentiability, nonlinearity and this 0,1 range gives us an amazing probability of return values that is a nice function.

However, this approach has its drawbacks. For instance, when you use backpropagation, you must back-propagate the derivative from your input back to its initial weights. You want to pass your regression or classification error in that final output value back through your whole neural network.

Therefore, you must derive your layers as well as update all weights. However, with sigmoid, there is an issue with a derivative. With sigmoid, the max value of the derivative is quite small just around 0.25. This means you can only pass a small fraction of your error to your previous neural network layers. This issue may cause your model to learn slow, so it needs more epochs and data. To solve this problem, you can use the tanh function.

TANH ACTIVATION FUNCTION

Tanh activation function, just like sigmoid, is differentiable and nonlinear. Tanh activation functions give output which is in the -1,1 range which is not as nice as 0,1 range. However, this is okay for neural network hidden layers. Tanh functions also have maxed derivative, which is good for our issue here as we can easily pass our error, which was not the case with sigmoid functions.

To use the tanh activation function, you must change the activation attribute of your dense layer.

```
model = Sequential( )
model . add ( Dense ( 512, activation = 'tanh', input _ shape = ( 784,)))
model . add ( Dense ( 10, activation = 'softmax' ))
```

Again, you can see that the structure of your neural network is the same. Now, train for five epochs.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [ ============================== ] - 5s - loss: 0.3333 - acc: 0.9006 -
val_loss: 0.2106 - val_acc: 0.9383
Epoch 2/5
60000/60000 [ ============================== ] - 3s - loss: 0.1754 - acc: 0.9489 -
val_loss: 0.1485 - val_acc: 0.9567
Epoch 3/5
60000/60000 [ ============================== ] - 3s - loss: 0.1165 - acc: 0.9657 -
val_loss: 0.1082 - val_acc: 0.9670
Epoch 4/5
60000/60000 [ ============================== ] - 3s - loss: 0.0843 - acc: 0.9750 -
val_loss: 0.0920 - val_acc: 0.9717
Epoch 5/5
60000/60000 [ ============================== ] - 3s - loss: 0.0653 - acc: 0.9806 -
val_loss: 0.0730 - val_acc: 0.9782

Test loss: 0.073
Test accuracy: 0.978
```

You can see that you improved your test accuracy by more than one percent just by using a different activation function. Now, you probably wonder, can you do better? Fortunately, you can thanks to the relu activation function.

RELU ACTIVATION FUNCTION

The range of relu activation functions is 0 to infinity. However, unlike tanh and sigmoid functions, relu is both differentiable at zero even though there are solutions to this.

The best thing about relu activation function is its gradient, which is always equal to one, so this way you can easily pass the maximum amount of the error during backpropagation. Now, train your model and see the results.

Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 5s - loss: 0.2553 - acc: 0.9263 - val_loss: 0.1505 - val_acc: 0.9516
Epoch 2/5
60000/60000 [==============================] - 3s - loss: 0.1041 - acc: 0.9693 - val_loss: 0.0920 - val_acc: 0.9719
Epoch 3/5
60000/60000 [==============================] - 3s - loss: 0.0690 - acc: 0.9790 - val_loss: 0.0833 - val_acc: 0.9744
Epoch 4/5
60000/60000 [==============================] - 4s - loss: 0.0493 - acc: 0.9844 - val_loss: 0.0715 - val_acc: 0.9781
Epoch 5/5
60000/60000 [==============================] - 3s - loss: 0.0376 - acc: 0.9885 - val_loss: 0.0645 - val_acc: 0.9823

Test loss: 0.064
Test accuracy: 0.982

Now, you go the best result of 98.2%. This is quite amazing, and you did not use any hidden layer.

It is very important to say there is no best activation function you can use. One may be better in some cases while another is better in other instances. Another important thing to say is that using different activation functions does not in any way affect what your neural network can learn and how fast.

CHAPTER 5: MNIST HANDWRITTEN RECOGNITION

In this section of the book, we are going to build a simple neural network in Keras and train it on a GPU-enabled server. This model will be able to recognize handwritten digits thanks to the MNIST dataset. As you already know, MNIST contains 70000 images, 10000 for testing and 60000 for training. All images are 28x28 pixels, centered to reduce preprocessing times.

To start, you must set your environment with Keras using Theano or TensorFlow as the backend. In this example, we are going to use the TensorFlow and Keras packages as shown below.

conda install - qy -c anaconda tensorflow - gpu h5py

pip install keras

These imports are quite standard. Once done, you must import Keras imports as follows. You will import plotting and array-handling. Once complete, you must keep your Keras backend TensorFlow quiet.

```
import numpy as np
import matplotlib
matplotlib . use ( 'agg' )
import matplotlib . pyplot as plt
import os
os . environ [ 'TF CPP MIN LOG LEVEL' ] =  '3'
from keras . datasets import mnist
from keras . models import Sequential, load _ model
from keras . layers . core import Dense, Dropout, Activation
from keras . utils import np _ utils
```

After that is done, Keras will import the dataset and build it on your neural network. The following step is to prepare the dataset we are going to use, MNIST. You must load the dataset using this very handy function that will split MNIST into test sets and train sets.

```
( X train, y train ), ( X test, y test ) = mnist . load _ data ()
```

The next step is to inspect several examples. Take into consideration that MNIST contains only grayscale images, so for more advanced datasets, we will

use RGB or three-color channels.

```
fig = plt . figure ()
for i in range(9):
  plt . subplot (3,3,i+1)
  plt . tight _ layout ()
  plt . imshow ( X _ train [i], cmap = 'gray', interpolation = 'none' )
  plt . title ( "Class {}". format( y _ train [i] ))
  plt . xticks ([])
  plt . yticks ([])
fig
```

Next, you begin to train your model to classify images. To do this, you must unroll the width height pixel format into one huge vector, your input vectors. Therefore, graph the distribution of your pixel values as follows.

```
fig = plt . figure ()
plt . subplot (2,1,1)
plt . imshow ( X _ train [0], cmap = ' gray ', interpolation =' none ' )
plt . title ( " Class {}". format( y _ train [0]))
plt . xticks ([])
plt . yticks ([])
plt . subplot (2,1,2)
plt . hist (X _ train [0] . reshape (784))
plt . title( " Pixel Value Distribution " )
fig
```

Just as expected, you get a pixel value ranging from zero to 255. In this case, the background majority is closer to zero while those pixels closer to 255 represent MNIST digits. To speed up the model training, you should normalize the input data. Normalizing your input data, you also reduce the chance of your model getting stuck in local optima as you are using stochastic gradient descent to find the optimal weights for your neural network.

The next step is reshaping your inputs to a single vector and normalizing the pixel value to be between zero and one. To do so, you must print the shape before you can normalize and reshape it.

```
Print ( " X _ train shape ", X _ train . shape )
Print ( " y _ train shape ", y _ train . shape )
Print ( " X _ test shape ", X _ test . shape )
print( " y _ test shape ", y _ test . shape )
```

After that, you must build your input vector from 28x28 pixels as seen below.

```
X _ train = X _ train . reshape (60000, 784)
X _ test = X _ test . reshape (10000, 784)
X _ train = X _ train . astype ('float32')
X _ test = X _ test . astype ('float32')
```

The next step is to normalize the data to boost your model training.

```
X _ train / = 255
X _ test / = 255
```

The following step is to print your final input shape, which is ready for training.

print(" Train matrix shape ", X _ train . shape)

print(" Test matrix shape ", X _ test . shape)

(' X _ train shape', (60000, 28, 28))

(' y _ train shape', (60000,))

(' X _ test shape', (10000, 28, 28))

(' y _ test shape', (10000,))

(' Train matrix shape ', (60000, 784))

(' Test matrix shape ', (10000, 784))

As you can see, Y in this training model holds integer values from zero to nine. Use it for model training.

print(np . unique(y _ train, return _ counts = True))

(array ([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8), array([5923, 6742, 5958, 6131, 5842, 5421, 5918, 6265, 5851, 5949]))

The next step is to encode your categories, digits from zero to nine using one-hot encoding. You will get the result that is a vector with a length equal to your number of categories. In addition, the vector you get is all zeros except in the middle position.

N _ classes = 10

Print (" ", y _ train .shape)

Y _ train = np utils . to categorical(y train, n classes)

Y _ test = np utils . to categorical(y _ test, n _ classes)

print(": ", Y _ train . shape)

The next step is to turn to Keras to build your neural network. At this point, your pixel vector serves as the input. There are two hidden 512-node layers as well. Therefore, there is model complexity you will use for recognizing digits. We must add another fully-connected layer for the ten different output classes due to multi-class classification. Further, we are going to use the sequential model. The first step is to stack more layers using the add argument.

When you add your first layer in the Keras sequential model, you must specify your input shape for Keras to create the proper matrices while the shape for other layers is inferred by Keras automatically. To introduce nonlinearities into your network and to evaluate it beyond capabilities of a basic perceptron, you must add activation functions to your hidden layers.

This differentiation for the model training using backpropagation is occurring behind the scenes. You must add a dropout model as the best way for preventing model over-fitting. You will use the softmax activation as the standard for every multi-class targets. When building your model, the first step is to build a linear stack of layers.

```
model = Sequential()
model . add ( Dense ( 512, input _ shape = ( 784 ,)))
model . add ( Activation( 'relu' ))
model . add ( Dropout (0.2))

model . add ( Dense(512 ))
model . add ( Activation(' relu '))
model . add ( Dropout(0.2 ))
```

model . add (Dense (10))

model . add (Activation (' softmax '))

Following that, you must compile your model using the compile argument. In this step, you must specify your objective or loss function. In this example, we are going to see categorical crossentropy, but you can use any other loss functions.

When it comes to the optimizers, in this example we are going to use the Adam optimizer with default settings. You can instantiate your optimizer and set parameters just before you use the model compile argument. You must choose which metrics you want to evaluate during model training and testing. You can have metrics displays during your testing and training stage if you like. To compile your model, do as follows.

Model . compile (loss = ' categorical_crossentropy ', metrics=['accuracy'], optimizer = 'adam')

Once you compile your model, you can move to model training. You must specify how many times you want to iterate your training set or epochs. You must specify how many samples you want to update to the model's batch size or weights.

Keep in mind that bigger the batch, the more stable stochastic gradient descent updates become. However, be aware of GPU memory limitation. In this example, we are going with a batch size of 8 and 128 epochs.

To handle your model training process correctly, you should graph the learning curve for your model looking only at the model accuracy and loss. Before you continue, you should save your model. Once completed, you get to work with your trained model and finally evaluate its performance. To save metrics run as shown below.

```
history = model . fit ( X train, Y train,
        batch _ size = 128, epochs = 8,
        verbose = 2,
        validation _ data = ( X test, Y test ))
```

Then, you must save your model.

```
Save _ dir = "/ results /"
Model _ name = 'keras_mnist.h5'
Model path = os . path . join( save dir, model _ name)
Model . save ( model _ path)
print( 'Saved trained model at %s ' % model _ path )
```

The next step is to plot the metrics.

```
fig = plt . figure ()
plt . subplot( 2,1,1 )
plt . plot( history . history[ 'acc' ])
plt . plot ( history . history[ 'val_acc' ])
plt . title ( 'model accuracy' )
plt . ylabel ( 'accuracy' )
plt . xlabel ( 'epoch' )
plt . legend ([ 'train', 'test'], loc = 'lower right')

plt . subplot (2,1,2)
plt . plot( history . history ['loss'])
plt . plot( history . history[ 'val_loss' ])
plt . title( 'model loss' )
plt . ylabel( 'loss' )
plt . xlabel( 'epoch' )
plt . legend([ 'train', 'test'], loc = 'upper right')
plt . tight _ layout ()
fig
```

You will notice that the loss on your training set is decreasing rapidly when it comes to the first two epochs. This means that your neural model is learning to classify handwritten digits quite fast. When it comes to the test set, the loss will not be decreasing as fast, but it will stay within the range of the training loss. This means that your model is able of generalizing well to data, which is unseen.

The following step is to evaluate your model's performance. In this step, you are going to see how well your model performs on the given test set. To assess your model, use model evaluates from the argument that compute any metric defined during the model compile process. In this example, the model accuracy is computed on the 10000 images of testing examples using the model weights given by our saved model.

```
Mnist model = load model
Loss _ and metrics = mnist model . evaluate( X test, Y test, verbose = 2 )
print( "Test Loss" , loss and metrics [0])
print( "Test Accuracy" , loss and metrics[1])
```

```
( 'Test Loss', 0.06264158328680787 )
( 'Test Accuracy', 0.98299999999999998 )
```

You will get this model accuracy that looks quite good. However, you should look at nine examples each so you evaluate both incorrectly and correctly classified examples. The first step is to load the model and create predictions on your test set.

```
Mnist model = load model
Predicted classes = mnist model . predict classes ( Xtest )
```

Then, see what you predicted correctly and incorrectly.

```
Correct indices = np . nonzero ( predictedclasses == y _ test)[0]
```

```
Incorrect _ indices = np . nonzero ( predicted classes ! = y test)[0]
Print ()
Print ( len ( correct _ indices )," classified correctly")
Print ( len(incorrect _ indices )," classified incorrectly")
```

The following step is to adapt the figure size to accommodate eighteen subplots as follows.

```
Plt . rcParams [' figure . figsize '] = (7,14)
Figure _ evaluation = plt . figure()
```

Then, you must plot nine correct and nine incorrect predictions.

```
(correct _ indices [:9]):
    Plt . subplot (6,3,i+1)
    Plt . imshow ( X _ test [correct] . reshape(28,28), cmap = 'gray', interpolation='none' )
    Plt . title ("Predicted {}, Class {}".format (predicted _ classes[correct], y _ test[correct]))
    Plt . xticks ([])
    Plt . yticks ([])
(incorrect _ indices[:9]):
    Plt . subplot(6,3,i+10)
    Plt . imshow( X _ test[incorrect].reshape(28,28), cmap ='gray', interpolation='none')
    Plt . title("Predicted {}, Class {}".format( predicted _ classes[incorrect], y _ test[incorrect]))
    Plt . xticks([])
    Plt . yticks([])
Figure _ evaluation
9696/10000 [ ============================== >.] - ETA: 0s()
```

(9830, ' classified correctly')

(170, ' classified incorrectly')

As you can see, these incorrect predictions are quite forgivable as in some cases it is hard for the human reader to recognize. In this section of the book, we used Keras with its backend TensorFlow on a GPU enabled server to train our neural network to recognize the handwritten digits in just under 20 seconds of overall training time.

CHAPTER 6: NEURAL NETWORK MODELS FOR MULTI-CLASS CLASSIFICATION PROBLEMS

As you already know, Keras is a highly powerful part of the Python library for deep learning, which wraps the efficient numerical libraries TensorFlow and Theano. In this section of the book, you are going to learn how to use Keras to develop and evaluate your neural network models you can use for assorted multi-class classification problems.

After that, you will know how to load data from CSV to Keras, you will how to prepare multi-class classification data for further modeling with your neural networks, and you will know how to evaluate your Keras neural network models using scikit-learn.

In this specific example, we are going to see one of the standard machine learning problems named iris flowers dataset. This problem is well-studied, so it is the best example to use when you want to practice more on neural networks as all four input variables are numeric meaning and they have the same scale represented in centimeters. In addition, every instance describes the properties of the flower measurements, so the output variables are very specific iris species.

This is a standard multi-class classification problem. This means there will be more than two classes you must predict, as there will be three flower species.

This is the best illustration to use when you want to practice on neural network models in Keras because these three class values require very specialized handling. Since this iris flower dataset is a very common and well-studied problem, expect to get your model accuracy somewhere between 95% to 97%. To start, you must download this iris flower dataset from the Machine Learning repository. Once downloaded, place it in your working directory. The next step is to import all classes and function. This includes both the data loading from pandas as well as data preparation. You need model evaluation from scikit-learn.

```
import numpy
import pandas
from keras . models import Sequential
from keras . layers import Dense
from keras . wrappers . scikit _ learn import KerasClassifier
from keras . utils import np_utils
from sklearn . model selection import cross val _ score
from sklearn . model _ selection import KFold
from sklearn . preprocessing import LabelEncoder
from sklearn . pipeline import Pipeline
```

The next step is to initialize a random number generator to a constant value of seven. This is very important, as you want to ensure that the results you get from this neural network model can in fact be achieved again.

This step ensures that the stochastic process of model training can be reproduced. Therefore, your next step is to fix random seed for reproducibility as seen below.

```
seed = 7
numpy . random . seed (seed)
```

Once fixed, you must load the dataset directly. You can do it directly as the output variable contains strings, so the best way is to load the data using pandas. In addition, you can split attributed into input variables as X and output variables as Y.

```
dataframe = pandas . read _ csv( "iris.csv", header= None )
dataset = dataframe . values
X = dataset [:,0:4] . astype (float)
Y = dataset[:,4]
```

ONE-HOT ENCODING

As already mentioned, in this example, the output variable contains three string values. Therefore, you must encode the output variables. When you model multi-class classification problems with neural networks, the best way is to reshape your output attributed from a vector which contains values for every class to a matrix that has a Boolean for every class value no matter given instance of the class value. This is called creating dummy variables or one-hot encoding of categorical variables.

For instance, in this problem here, there are three class values named Iris-versicolor, Iris-setosa and Iris-virginica.

Iris - setosa

Iris - versicolor

Iris - virginica

In this case, you can turn this into a single one-hot encoded binary matrix for every data instance, which would like as shown below.

Iris-setosa, Iris-versicolor, Iris-virginica

1, 0, 0

0, 1, 0

0, 0, 1

You can do this by just encoding the strings to integers with the scikit-learn class

Label Encoder. Once completed, you can convert the integer vector to a single one-hot encoding using the function to categorical in Keras as demonstrated below.

encoder = LabelEncoder ()

encoder . fit (Y)

encoded _ Y = encoder . transform (Y)

dummy _ y = np utils . to categorical(encoded _ Y)

DEFINING NEURAL NETWORK MODELS WITH SCIKIT-LEARN

Once done with one-hot encoding, you must define your neural network model. The Keras library comes with wrapper classes, which allow you to use your neural network models you created in Keras in scikit-learn. Plus, there is a KerasClassifier class that can be used as scikit-learn estimator.

This KerasClassifier takes the name of the functions as an argument. Consider that this function must return to your neural network model which is ready for training. Further, we are going to create a function for this iris classification problem.

Once you run the code, you will create a simple, fully-connected neural network containing one hidden layer with eight neurons. This hidden layer will use a rectified activation function that is a very good practice. Since we already used one-hot encoding on this iris dataset, your output layers must create three output values for each class. Once done, the output value with the biggest value becomes the class predicted by your neural network model as follows.

4 inputs -> [8 hidden nodes] -> 3 outputs

One thing should be noted. We used a softmax activation function in our output layer to ensure that the output values of our model are in the range of zero and one, so they may be used as our predicted probabilities.

Finally, we must use the efficient Adam gradient descent optimization model alongside logarithmic loss function represented as the categorical crossentropy argument in Keras. Therefore, the next step is to define your baseline model. Once completed, you must create it and compile it as below.

```
def baseline _ model ()
model = Sequential ()
model . add ( Dense(8, input_dim=4, activation= 'relu' ))
model . add (Dense(3, activation='softmax'))
model . ( loss ='categorical_crossentropy', optimizer ='adam', metrics= ['accuracy'])
return model
```

After that, you can finally create your KerasClassifier in scikit-learn. You can pass arguments as well during the construction of your KerasClassifier, which will be quickly passed on to your fit function, which you will use for training your neural network model. Following, we are going to pass the number of epochs as two hundred and batch size as 5 to use them during model training. Bear in mind that debugging is also turned off as we set verbose to zero.

```
estimator = KerasClassifier ( build fn = baseline model, epochs = 200, batch _ size = 5, verbose = 0)
```

EVALUATING MODELS WITH K-FOLD CROSS VALIDATION

Once finished with the previous step, you must evaluate your neural network model on your training data. The powerful scikit-learn has excellent capability of evaluating neural network models using several different techniques. The best way for evaluating your neural network models is using k-fold cross validation.

Using k-fold cross validation, you can evaluate your model on your dataset using a ten-fold cross validation argument or k-fold. The process of evaluating your model will take about ten seconds.

When finished, your model will return as an object which describes the evaluation of the ten constructed models for each of the splits in the dataset as shown below.

results = cross _ val score (estimator, X, dummy y, cv = kfold)

*print("Baseline: %.2f%% (%.2f%%)" % (results.mean ()*100, results.std()* 100))*

Once completed, you will see that the results are summarized as both the standard and mean deviation of your neural network model accuracy on the dataset we used.

This is a very reasonable estimation of the overall performance of your neural network model on this unseen data. This is well within the realm of known

results for this specific problem as you get accuracy as seen below.

Accuracy: 97.33% (4.42%)

CHAPTER 7: RECURRENT NEURAL NETWORKS

In this last section of the book, you are going to learn how to create recurrent neural networks in Keras. Recurrent neural networks are a class of neural network models, which exploit the sequential nature of their input. Such inputs can be speech, text, time series, and everything else where the occurrence of an element in the sequence is dependent on the elements, which appeared before it.

An RNN model can be thought of as a graph of recurrent neural network cells where every cell performs the same operation on each element in the sequence.

Recurrent neural networks are very flexible, so they have been used to solve diverse problems like language modeling, speech recognition, sentiment analysis, machine translation and image captioning, to name a few.

Recurrent neural networks can be readily adapted to many kinds of problems just by rearranging the way the cells are situated in the graphs. In this section of the book, you are going to learn more about LSTM or long short-term memory and GRU or gated recurrent unit models, about their powers and their limitations.

Both GRU and LSTM are drop-in replacements for the basic recurrent neural network cell, so just by replacing the recurrent neural network cell with one of these two variations you can get a major performance boost in your network.

While GRU and LSTM are not the only variants, they have proven to be the most efficient for solving most sequence problems.

SEQUENCE CLASSIFICATION WITH LSTM RECURRENT NEURAL NETWORKS

Sequence classification is a common predictive modeling problem in which you have a sequence of inputs placed over time or space, and your task is to predict a specific category for that sequence. A powerful type of neural network model created to handle problems like this is a LSTM recurrent neural network.

The long short-term memory is a type of recurrent neural network ordinarily used in deep learning problems due to its large architectures which can be successfully trained. In this section, you are going to learn sequence classification in Keras using LSTM recurrent neural networks.

What makes this problem difficult is that the specific sequences can vary in length, they may sometimes contain the very large vocabulary of their input symbols and they may require your neural network model to learn that long-term context of diverse dependencies existing between different symbols in your input sentence.

The problem we are going to solve is the IMDB movie sentiment classification problem. Each movie review on the IMDB is a variable sequence of words while each sentiment of every movie review must be classified. We will use the IMDB dataset that contains 25,000 movie reviews both good or bad for testing and training. The problem here is to determine whether the movie has a negative or positive sentiment.

Keras comes with built in access to the IMDB dataset. To load it, use IMDB load data function. Once loaded, you can use it for your deep learning models. The words here have been replaced by integers, which indicate the specific ordered frequency of each word in the IMDB dataset while the sentences in each movie review are comprised of a certain sequence of integers.

WORD EMBEDDING

Our first move is to map each movie review into a real vector domain. This is a very popular technique used with text named word embedding. In this technique, words are encoded as real-valued vectors in a high dimensional space in which the similarity between words, in terms of their meaning, translates to the closeness of that vector space.

Keras is good for this as it provides a highly effectively and convenient way for converting positive integer word representations into a word embedding using Embedding layers. Therefore, we are going to map each word onto a thirty-two-length real-valued vector. In addition, we are going to limit the total number of words we are interested in neural network modeling to the five thousand most frequent words and zero out the remaining.

Moreover, we are going to constrain each movie review to be five hundred words as we truncate long movie reviews and pad those shorter reviews with zero values. The first step is to prepare and model data. Once done, you are ready to create your LSTM model, which will classify the sentiment of movie reviews.

Your first step is to quickly develop a basic LSTM for this IMDB problem. Start with importing functions and classes, which are required for this model. Then, initialize the random number generator to a constant value to make sure you can effortlessly reproduce the results you get.

import numpy

from keras . datasets import imdb

from keras . models import Sequential

from keras . layers import Dense

from keras . layers import LSTM

from keras . layers.embeddings import Embedding

from keras . preprocessing import sequence

numpy . random . seed (7)

Once done, you must load the IMDB dataset. Additionally, you will constrain the dataset to the top five thousand words. You must split the dataset into test and train sets.

Top _ words = 5000

(X train, y train), (X test, y test) = imdb . load _ data (num words = top words)

The following step is truncating and padding your input sequences, so they are same in the length. Your model will learn these zero values that carry no information, so same length vectors are required for computation here.

Max reviewlength = 500

X train = sequence.padsequences(X train, maxlen = max review _ length)

X _ test = sequence . pad_sequences(X test, maxlen= max review _ length)

Once completed, you must define, compile and finally fit your LSTM model.

The first layer in your embedded layer uses thirty-two-length vectors, which represent every word. The following layer is your LSTM layer that has hundred smart or memory units. Furthermore, you must use a dense output layer containing a sigmoid activation function and a single neuron to make zero and one predictions for your two classes, good or bad reviews, in the problem.

Since this is a binary classification problem, you must use log loss as your loss functions in addition to the efficient Adam optimizer. Your models will be fit for only two epochs, so it will quickly over-fit this problem. To space out weight updates, you will use a big batch size of sixty-four movie reviews.

```
Embedding vecor length = 32
model = Sequential()
model . add ( Embedding (top_words, embedding vecor length, input length = max review _ length))
model . add ( LSTM(100))
model . add ( Dense(1, activation = 'sigmoid'))
model . compile (loss='binary_crossentropy', optimizer ='adam', metrics = ['accuracy'])
print ( model . summary ())
model . fit ( X train, y train, validation _ data = ( X test, y test), epochs = 3, batch _ size = 64)
```

The next step is to estimate the performance of your model on a few unseen movie reviews as follows.

```
scores = model . evaluate(X test, y test, verbose = 0)
print( "Accuracy: %.2f%%" % ( scores [1]*100))
```

Running this code, you will get as indicated below.

Epoch 1/3

16750/16750 [==============================] - 107s - loss: 0.5570 - acc: 0.7149

Epoch 2/3

16750/16750 [==============================] - 107s - loss: 0.3530 - acc: 0.8577

Epoch 3/3

16750/16750 [==============================] - 107s - loss: 0.2559 - acc: 0.9019

Accuracy: 86.79%

APPLYING DROPOUT

This very simple LSTM model with little tuning provides great results on this IMDB dataset problem. Use this model as a template which you can apply to a variety of LSTM neural networks to your own sequence classification problems.

Recurrent neural networks like LSTM frequently come with over-fitting problems you can solve by applying the dropout Keras layer between layers. You just add new layers between your embedding and LSTM layers and between your LSTM and Dense output layers as follows.

```
model = Sequential ()
model . add ( Embedding (top_words, embedding vecor length, input length = max review _ length))
model . add (Dropout(0.2))
model . add (LSTM(100))
model . add (Dropout(0.2))
model . add (Dense(1, activation ='sigmoid'))
```

Running this you will get following result.

```
Epoch 1/3
16750/16750 [ ============================== ] - 112s - loss: 0.6623 - acc: 0.5935
Epoch 2/3
16750/16750 [ ============================== ] - 113s - loss: 0.5159 - acc: 0.7484
Epoch 3/3
16750/16750 [ ============================== ] - 113s - loss: 0.4502 - acc: 0.7981
Accuracy: 82.82%
```

As you can see, the dropout layer has an impact on training with a lower final accuracy and slower trend in convergence. This LSTM model probably could use several more epochs of training for better skill. Dropout can also be applied to the recurrent connections of the memory units with the LSTM separately and precisely.

Keras comes with amazing capability on the LSTM layers. You can use the dropout function for configuring your input dropout and your recurrent dropout. You can modify the code and add dropout to your recurrent connections and to the input as illustrated below.

```
model = Sequential()
model . add ( Embedding (top_words, embedding_vecor_ length, input length = maxreview_length))
model . add (LSTM(100, dropout = 0.2, recurrent _ dropout=0.2))
model . add (Dense(1, activation = 'sigmoid'))
```

You can see that this LSTM specific dropout has more effect on the layer-wise dropout and on the convergence of your network. Dropout is a very powerful technique you should use for combating over-fitting issues in your LSTM models. Make sure you use both methods even though you may get better results when using this gate-specific dropout method.

NATURAL LANGUAGE PROCESSING WITH RECURRENT NEURAL NETWORKS

In this section of the book, we are going to solve a natural language processing problem using recurrent neural networks in Keras. This natural language processing problem aims to extract the meaning of speech utterances. We are going to break this problem into solvable practical issues of understating the speaker in a limited context. Here, we want to identify the intent of a speaker asking for info about flights.

We are going to use Airline Travel Information System or ATIS. This dataset was obtained by DARPA back in the early 90s. The dataset consists of spoken queries on numerous flights. ATIS contains 4,978 sentences and 56,590 words both in the test and train set. The number of classes contained in 128. Our approach here is to use recurrent neural networks and word embedding.

As you already know, word embedding maps words to vectors in a high-dimensional space. The word embedding when learned right can learn syntactic and semantic information of the words in this space. This embedding space will be learned by your model that you define later.

For this problem, a convolutional layer can do great when it comes to pooling local information, but they are not capable of capturing the real data sequentially, so we are going to use recurrent neural networks which will help us tackle this consecutive information as natural language.

A recurrent neural network model has such a memory that stores the summary of countless sequences the model has seen before.

This means you can use recurrent neural networks to solve complex word tagging problems like POS or part of speech tagging or slot filling as in this problem.

For this problem, you must pass the word embeddings sequence as the input of your recurrent neural network.

As you are going to use IOB representation for your labels, it is necessary to calculate the scores of your model. Therefore, you will run code as shown below for your score calculation. Prior to that however, you must download the corresponding ATIS file.

git clone https: //github.com/chsasank / ATIS.keras.git

cd ATIS.keras

I recommend you to use Jupyter Notebook. After that, you must load your data using data load atisfull argument. Keras will download the data the first time you run it.

Labels and words are encoded as indexes to your dataset vocabulary and vocabulary is stored in labels2idx.

import numpy as np

import data.load

train _ set, valid _ set, dicts = data . load. atisfull() w2idx, labels2idx = dicts ['words2idx'], dicts['labels2idx']

train _ x, , train label = train _ set val _ x, , val label = valid _ set The next step is to create an index to label and word dicts as seen below.

idx2w = { w2idx[k]:k for k in w2idx }

idx2la = { labels2idx[k]:k for k in labels2idx }

Then, create conlleval script as follows.

Words _ train = [list(map (lambda x: idx2w[x], w)) for w in train _ x]

labels_train = [list(map (lambda x: idx2la[x], y)) for y in train _ label]

words_val = [list(map (lambda x: idx2w[x], w)) for w in val _ x]

labels_val = [list (map(lambda x: idx2la[x], y)) for y in val _ label]

n _ classes = len (idx2la)

n _ vocab = len (idx2w)

The next step is to print an example label and sentence.

print("Example sentence : {}".format(words _ train [0]))

print("Encoded form: {}".format(train _ x[0]))

print( )

print("It's label : {}". Format (labels_train[0]))

print("Encoded form: {}".format (train_label[0]))

This is what you get.

Example sentence : [...]

Encoded form: [232 542 502 196 208 77 62 10 35 40 58 234 137 62 11 234 481 321]

It's label : [...]

Encoded form: [126 126 126 126 126 48 126 35 99 126 126 126 78 126 14 126 126 12]

The next step is to define your Keras model. Keras comes with built embedding layer you can use for word embeddings. It will expect integer indices.

You also must use TimeDistributed argument to pass the ouput of your recurrent neural network at each time step to a fully connected layers. If you do not perform this step, your output at the time of the final step will be passed on your next layer.

from keras . models import Sequential

from keras . layers . embeddings import Embedding

from keras . layers . recurrent import SimpleRNN

from keras . layers . core import Dense, Dropout

from keras . layers . wrappers import TimeDistributed

from keras . layers import Convolution1D

model = Sequential( )

model . add (Embedding(n_vocab,100)) model . add (Dropout(0.25))

model . add (SimpleRNN(100, return _ sequences = True)) model . add (TimeDistributed(Dense(n _ classes, activation = 'softmax'))) model . compile('rmsprop', 'categorical _ crossentropy') The next

step is to train your model. You will pass every sentence as a batch to your model. You cannot use model fit argument as it expects all included sentences to be the same size. Therefore, you are going to use model train on batch argument.

```
import progressbar

n _ epochs = 30

print ( "Training epoch {}".format(i))


bar = progressbar . ProgressBar( max value = len( train x))

label = train label [n batch]
```

Then, you must make labels one-hot. When that step is finished, you must make your model view each sentence as a batch as indicated below.

```
label = np . eye (n _ classes)[ labe l][ np . newaxis,: ]

sent = sent[ np . newaxis ,:]

model . train on batch( sent, label)
```

To measure the accuracy of your model, you are going to use the model predict on batch argument alongside metrics accuracy conlleval argument.

```
from metrics . accuracy import conlleval

labels pred val = [ ]

bar = progressbar . ProgressBar (max _ value = len (val _ x))

for n _ batch, sent in bar ( enumerate( val _ x)):

    label = val label [n batch]

    label = np . eye(n_classes)[ label][np . newaxis,:]

    sent = sent[np . newaxis,:]

    pred = model . predict on batch (sent)
```

```
    pred = np . argmax ( pred, -1)[0]
    labels pred val . append(pred)
labels pred val = [ list ( map( lambda x: idx2la[ x ], y )) \
                        for y in labels pred val]
con _ dict = conlleval( labels pred val, labels _ val,
                    words_val, 'measure.txt' )
print(' Precision = {}, Recall = {}, F1 = {}'.format (
        con dict ['r'], con dict ['p'], con _ dict[ 'f1' ]))
```

With this model, you should get around ninety-two F1 Score. One drawback on this model is that there is no lookahead. You can handily implement it by adding a convolutional layer before recurrent neural network layers and just after word embeddings as follows.

```
model = Sequential( )
model . add ( Embedding(n _ vocab,100))
model . add ( Convolution1D (128, 5, border _ mode = 'same', activation = ' relu ' ))
model . add ( Dropout (0.25))
model . add ( GRU(100, return _ sequences = True))
model . add ( TimeDistributed ( Dense(n _ classes, activation='softmax' )))
model . compile( 'rmsprop', 'categorical _ crossentropy')
```

With this greatly improved model, you should get around ninety-four F1 Score. To improve your model even further, you can use other word embedding corpuses like Wikipedia. You can try other recurrent neural network variants like GRU and LSTM that allow more experimentation.

LAST WORDS

Deep learning is a new area of broader machine learning that has been introduced with the main goal of moving machine learning closer to artificial intelligence, which was one of its original goals. If you want to break deeper into artificial intelligence, first, you need to focus on deep learning and its powers. Deep learning arguably is one of the most highly sought tech skills.

This book will help you become good at deep learning basics. The book will help you start your deep learning journey properly. Since you are done with the reading part, you know a lot about neural networks models, how to build them and how to solve different deep learning problems like natural language processing and speech recognition. Therefore, you can focus on more advanced deep learning problems in the future.

In the book, you surveyed several neural network models and their applications to the real-world problems. You can use this knowledge in solving your own deep learning tasks as you build your own neural network models using Keras. One thing is for sure, you should take advantage of the knowledge you gained through the book and focus on more complex deep learning problems.

Deep learning is the only field of AI that went viral and its future looks very bright. Therefore, you should not stop here. You should focus on improving your skill and gaining more knowledge. Machine learning already plays a massive part in your everyday life and deep learning is not far away from becoming a

larger part of modern society as well.

Machine learning was just the beginning as more and more tech companies like Microsoft, Google and Facebook spend millions on deep learning and advanced neural networks research as computers get smarter every day.

However, deep learning is not about self-aware machines. It is about how ingenious neural network models and code are giving machines the ability to do things we previously thought impossible. Therefore, deep learning does concern our future. Let the book be your guide into this world, but do not stop here and make sure you take a step further by learning something new every day.