FUNDAMENTALS OF DEEP LEARNING

Chapter 1

What will we cover

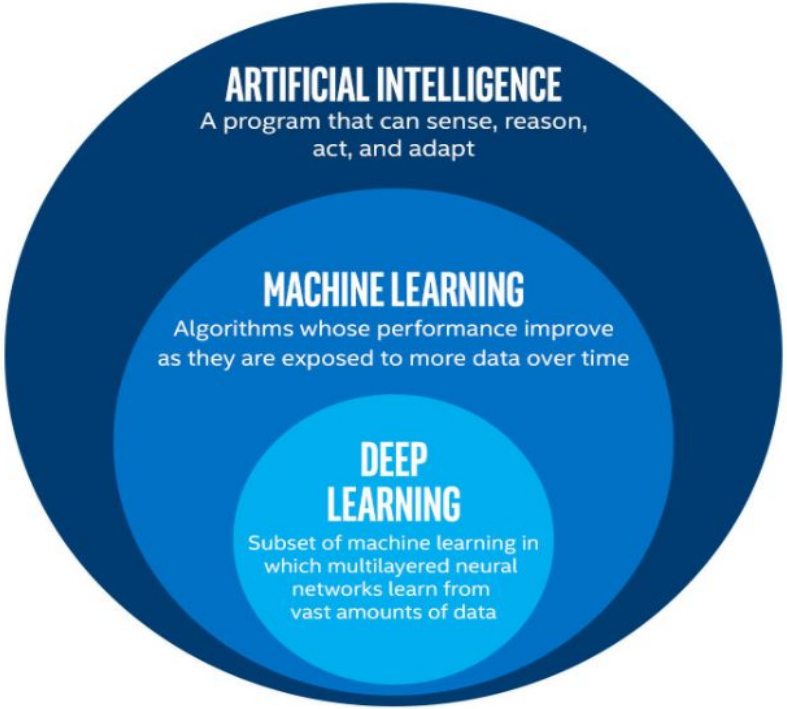
- What is Machine Learning
- Fundamental concepts involved in Machine Learning
- Four Branches of Machine Learning
- What is Deep Learning
- How it works
- What it can achieve

Text Book / Reference Book

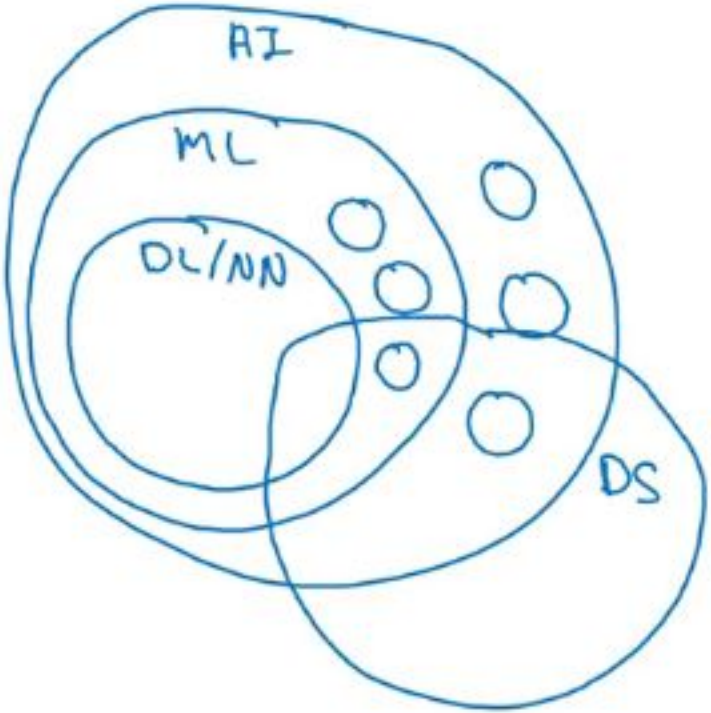
- Deep Learning with Python, Authored by FRANÇOIS CHOLLET

CHAPTER-1

AI, Machine Learning & Deep Learning



AI, Machine Learning & Deep Learning



ARTIFICIAL INTELLIGENCE

Artificial Intelligence

Idea of AI was born when scientists started to think / program computers to do the tasks only a human can do. For a long time Symbolic AI ruled the world in which we maintain a large set of rules. Symbolic AI had certain limitations in solving perception problems, like recognizing / tagging an image, translating a language to another language, etc.

How good an AI algorithm is (Turing Test)

MACHINE LEARNING

Machine Learning

The frustration of crafting hard coded rules made the scientists to think what if a program can infer the rules to describe the answers / results by itself. This thought pioneered the field of Machine Learning.

Classical Programs

Machine Learning

Essential Things in Machine Learning

For machine learning, we need three things:

- Input data points
- Examples of the expected output
- A way to measure how good an algorithm is doing

In simple words Machine Learning is to learn useful representations of the input data (representations that get us closer to the expected output)

DEEP LEARNING

Deep Learning

Deep learning (DL) is essentially a subset of ML that extends ML capabilities across multilayered neural networks to go beyond just categorizing data. DL can actually learn, self-train, essentially from massive amounts of data. With DL, it's possible to combine the unique ability of computers to process massive amounts of information quickly, with the human-like ability to take in, categorize, learn, and adapt.

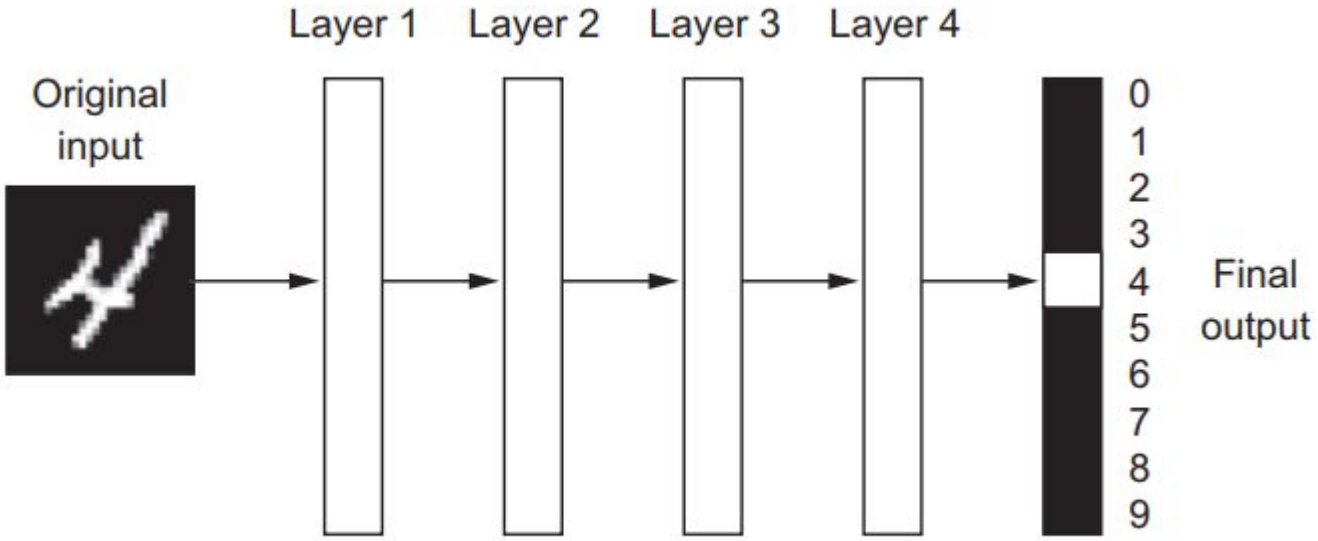
Reference:
https://www.prowesscorp.com/whats-the-difference-between-artificial-intelligence-ai-machine-learning-and-deep-learning/

Deep Learning

In deep learning, these layered representations are (almost always) learned via models called neural networks, structured in literal layers stacked on top of each other.

The term neural network is a reference to neurobiology, but although some of the central concepts in deep learning were developed in part by drawing inspiration from our understanding of the brain, deep-learning models are not models of the brain.

Neural Network for Handwriting Recognition



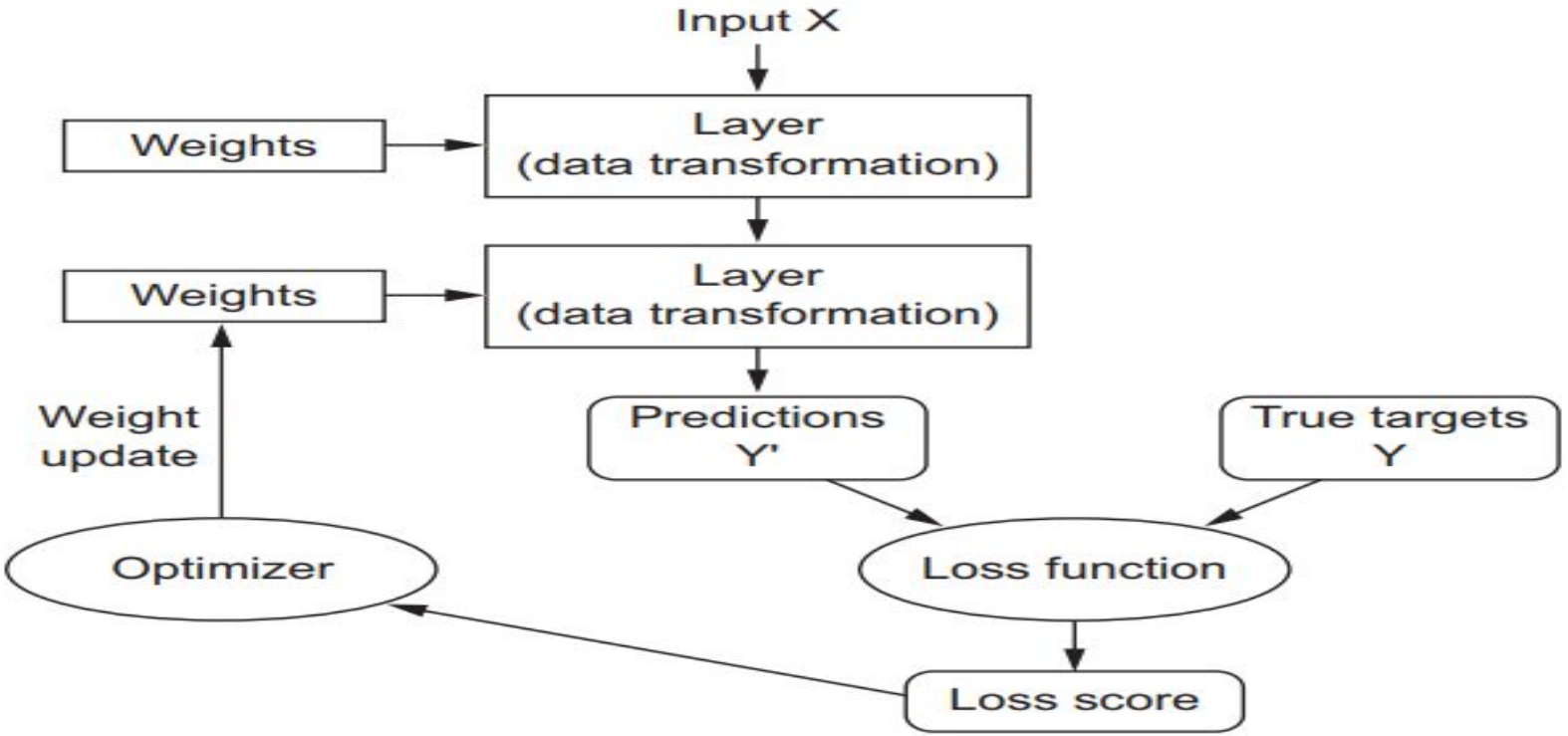
So that’s what deep learning is, technically:

A multistage way to learn data representations. It’s a simple idea but, as it turns out, very simple mechanisms, sufficiently scaled, can end up looking like magic.

Understanding How Deep Learning Works

$$Y = WX + B$$

Understanding How Deep Learning Works



What deep learning has achieved so far

In particular, deep learning has achieved the following breakthroughs, all in historically difficult areas of machine learning:

- Near-human-level image classification
- Near-human-level speech recognition
- Near-human-level handwriting transcription
- Improved machine translation
- Improved text-to-speech conversion
- Digital assistants such as Google Now and Amazon Alexa

What deep learning has achieved so far

- Near-human-level autonomous driving
- Improved ad targeting, as used by Google, Baidu, and Bing
- Improved search results on the web
- Ability to answer natural-language questions
- Superhuman Go playing

Is this an start towards Terminator Robots

BRIEF HISTORY OF MACHINE LEARNING

Probabilistic Modeling

Probabilistic modeling is the application of the principles of statistics to data analysis. It was one of the earliest forms of machine learning, and it’s still widely used to this day. One of the best-known algorithms in this category is the Naive Bayes algorithm.

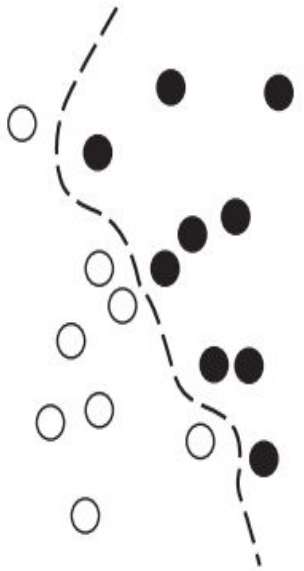
Early Neural Networks

Although the core ideas of neural networks were investigated in toy forms as early as the 1950s, the approach took decades to get started. For a long time, the missing piece was an efficient way to train large neural networks.

Kernel Methods

Kernel methods are a group of classification algorithms, the best known of which is the support vector machine (SVM). SVMs aim at solving classification problems by finding good decision boundaries between two sets of points belonging to two different categories.

A decision boundary can be thought of as a line or surface separating your training data into two spaces corresponding to two categories. To classify new data points, you just need to check which side of the decision boundary they fall on.



DECISION TREE, RANDOM FOREST AND BOOSTING MACHINES

Decision Trees

A decision tree is a hierarchical model in which every node splits the samples into branches against a rule. Every leaf of the tree is the label / prediction of the model. Before the rise of Deep learning; decision trees were the most popular technique and preferred choice in ML practitioners and researchers. They are able to capture linear and nonlinear relations in data. Another name for Decision Tree is CART (Classification And Regression Tree).

Decision Trees Example

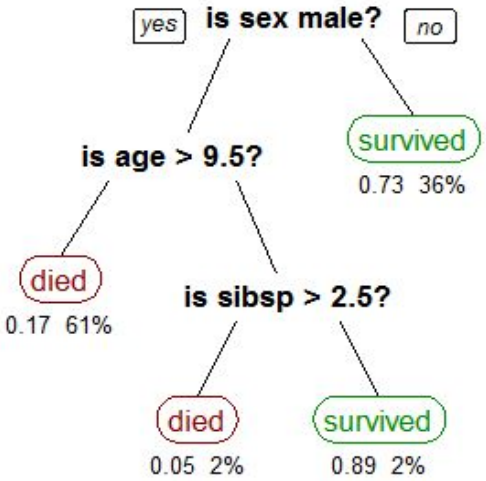
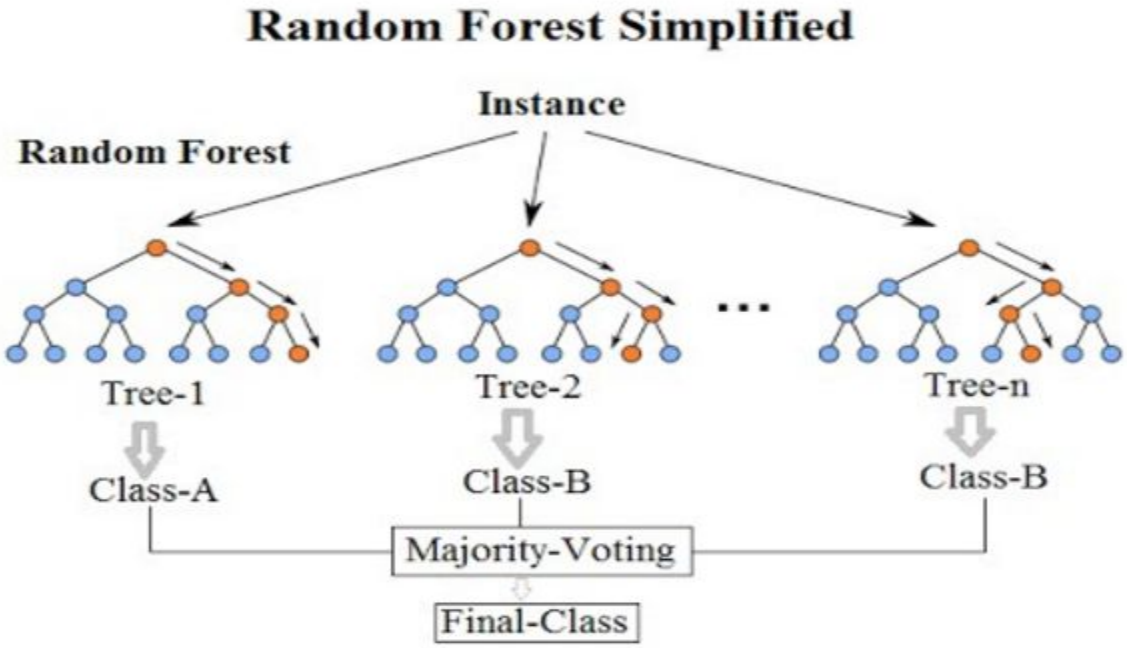


Image taken from wikipedia

Random Forest

Random forest is a technique in which a large number of CARTs (decision trees) are used to predict the outcome and a voting node is used to declare the final label on majority of votes from individual trees.

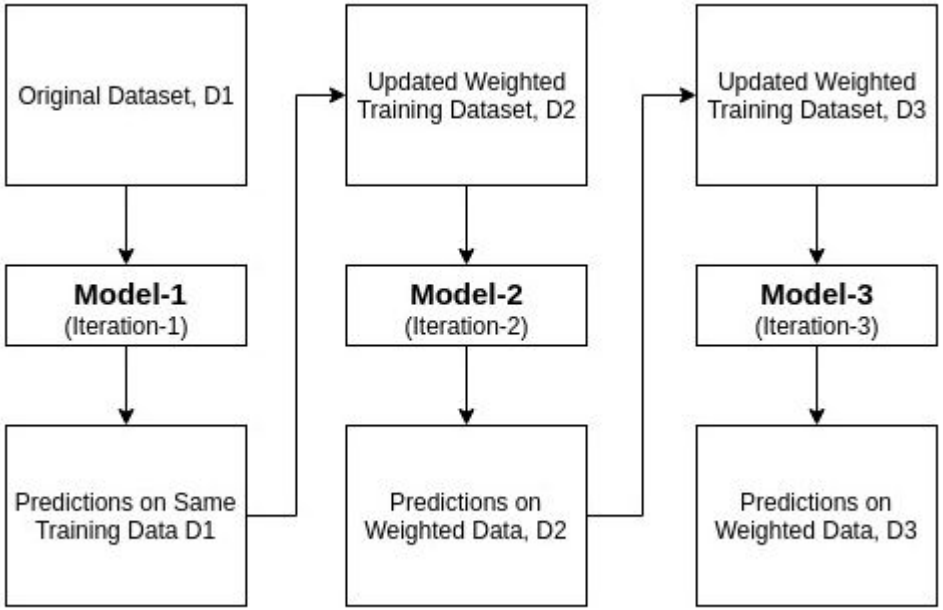
Random Forest Example



Boosting Machines

Boosting Machines are algorithms in which a large number of weak learners (Decision Trees) are used sequentially in such a way that every learner reduces the error of its predecessor's prediction. Two mostly used Boosting Machines are Ada Boost and Gradient Boost.

Boosting Machine Illustration



BACK TO DEEP LEARNING

What Makes Deep Learning Different

- It offers better performance on many problems with higher accuracy than classical techniques
- Makes Problem solving much easier by automating a very crucial and time consuming step in Classical Machine Learning techniques that is Feature Engineering.

What Makes Deep Learning Different

The two essential characteristics of how deep learning learns from data are

- The incremental, layer-by-layer way in which increasingly complex representations are developed
- These intermediate incremental representations are learned jointly

Each layer being updated to follow both the representational needs of the layer above and the needs of the layer below

Why Deep Learning? Why Now?

The two key ideas of deep learning for computer vision—convolutional neural networks and backpropagation were already well understood in 1989. The Long Short Term Memory (LSTM) algorithm, which is fundamental to deep learning for time series, was developed in 1997 and has barely changed since. So why did deep learning only take off after 2012?

What changed in these two decades?

Why Deep Learning? Why Now?

In general, these three technical forces are driving advances in machine learning:

- Hardware
- Data
- Algorithmic advances

Hardware

In past few years introduction of GPU and vendor libraries to compute complex tasks over GPU made the deep learning shine as complex tasks on a large amount of data are solved in a considerably small time. During the last year Google has introduced TPUs which are specifically designed for Deep Learning tasks and are even 10x faster than a GPU.

Data

When it comes to data, in addition to the exponential progress in storage hardware over the past 20 years (following Moore's law), the game changer has been the rise of the internet, making it feasible to collect and distribute very large datasets for machine learning. Today, large companies work with image datasets, video datasets, and natural-language datasets that couldn't have been collected without the internet. User-generated image tags on Flickr, for instance, have been a treasure trove of data for computer vision. So are YouTube videos. And Wikipedia is a key dataset for natural-language processing.

Algorithms

In addition to hardware and data, until the late 2000s, we were missing a reliable way to train very deep neural networks. As a result, neural networks were still fairly shallow, using only one or two layers of representations; thus, they weren't able to shine against more-refined shallow methods such as SVMs and random forests. The key issue was that of gradient propagation through deep stacks of layers. The feedback signal used to train neural networks would fade away as the number of layers increased.

A NEW WAVE OF INVESTMENT

A New Wave Of Investment

- AI and machine learning have the potential to create an additional $2.6T in value by 2020 in Marketing and Sales, and up to $2T in manufacturing and supply chain planning.
- Gartner predicts the business value created by AI will reach $3.9T in 2022.
- IDC predicts worldwide spending on cognitive and Artificial Intelligence systems will reach $77.6B in 2022.

Reference:

https://www.forbes.com/sites/louiscolumbus/2019/03/27/roundup-of-machine-learning-forecasts-and-market-estimates-2019/#399b12c07695

The Democratization Of Deep Learning

Introduction of new tools for languages that support Deep Learning made it to approachable to a common developer with the knowledge of high level scripting languages like Python.

Will it last?

Deep learning has several properties that justify its status as an AI revolution, and it's here to stay. We may not be using neural networks two decades from now, but whatever we use will directly inherit from modern deep learning and its core concepts.

These important properties can be broadly sorted into three categories:

- Simplicity
- Scalability
- Versatility and reusability

Simplicity

Deep learning removes the need for feature engineering, replacing complex, brittle, engineering-heavy pipelines with simple, end-to-end trainable models that are typically built using only five or six different tensor operations

Scalability

Deep learning is highly amenable to parallelization on GPUs or TPUs, so it can take full advantage of Moore's law. In addition, deep-learning models are trained by iterating over small batches of data, allowing them to be trained on datasets of arbitrary size. (The only bottleneck is the amount of parallel computational power available, which, thanks to Moore's law, is a fast moving barrier.)

Versatility And Reusability

Unlike many prior machine-learning approaches, deep-learning models can be trained on additional data without restarting from scratch, making them viable for continuous online learning an important property for very large production models. Furthermore, trained deep-learning models are repurposable and thus reusable: for instance, it’s possible to take a deep learning model trained for image classification and drop it into a video processing pipeline. This allows us to reinvest previous work into increasingly complex and powerful models. This also makes deep learning applicable to fairly small datasets.

THE MATHEMATICAL BUILDING BLOCKS OF NEURAL NETWORKS

Chapter 2

What will we cover

- A basic example of Neural Network
- What is a Tensor
- Tensor Operations
- How Neural Networks Learn
- Backpropagation
- Gradient Descent

Classes And Labels

In machine learning, a category in a classification problem is called a class. Data points are called samples. The class associated with a specific sample is called a label.

Classification Problem

A classification problem is when the output variable is a category, such as “red” or “blue” or “disease” and “no disease”.

Reference: https://www.geeksforgeeks.org/regression-classification-supervised-machine-learning/

INTRODUCTION OF GOOGLE COLAB (DEMO)

SETTING UP LOCAL ENVIRONMENT (DEMO)

Testing Environment

```
import tensorflow as tf

print(tf.__version__)

# should print the version of tensorflow module
```

BASIC EXAMPLE OF NEURAL NETWORK

Fashion mnist dataset

We will be using MNIST-like fashion product database, consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Each training and test example is assigned to one of the following labels:

Label	Description	Label	Description
0	T-shirt/top	5	Sandal
1	Trouser	6	Shirt
2	Pullover	7	Sneaker
3	Dress	8	Bag
4	Coat	9	Ankle boot

Fashion MNIST Data Loading

The Fashion MNIST data is available directly in the tf.keras datasets API. We can load it like this:

```
mnist = tf.keras.datasets.fashion_mnist

(training_images, training_labels), (test_images, test_labels) = mnist.load_data()
```

Data Normalization

In neural networks all input parameters are normalized to have a value between 0 (Zero) and 1 (One). To do so we will divide every pixel value from 255, using our knowledge of numpy we can do it with a single line in Python:

```
training_images  = training_images / 255.0

test_images = test_images / 255.0
```

Model Creation

```
model =
tf.keras.models.Sequential([tf.keras.layers.Flatten(),
tf.keras.layers.Dense(128, activation=tf.nn.relu),
tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

Here we have created a neural network of 3 layers:

Flatten: Serves as input layer and flattens the rectangular pic array in 1d array

Dense: Commonly known as hidden layer contains 128 neurons

Dense: Serves as output layer contains 10 neurons as we have 10 classes in dataset

Training Model

Now it's time to train the model, observe the accuracy and loss at each epoch:

```
model.compile(optimizer = tf.keras.optimizers.Adam(),
              loss = 'sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(training_images, training_labels, epochs=10)
```

Evaluating Model

Once the model is trained we can measure it’s accuracy over the test set which has not been observed by model. Again observe the accuracy and loss of the model predictions:

```
model.evaluate(test_images, test_labels)
```

Basic Example Of Neural Network

```
import tensorflow as tf

mnist = tf.keras.datasets.fashion_mnist

(training_images, training_labels), (test_images, test_labels) = mnist.load_data()

training_images  = training_images / 255.0

test_images = test_images / 255.0

model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
tf.keras.layers.Dense(512, activation=tf.nn.relu),
tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

model.compile(optimizer = tf.keras.optimizers.Adam(), loss =
'sparse_categorical_crossentropy', metrics = ['accuracy'])

model.fit(training_images, training_labels, epochs=5, batch_size=128)

model.evaluate(test_images, test_labels)
```

Data Representations For Neural Networks

Basic building blocks for data passing into neural networks are numpy arrays, also called tensors. At its core, a tensor is a container for data almost always numerical data. In other words it’s a container for numbers. Tensors are a generalization of matrices to an arbitrary number of dimensions (note that in the context of tensors, a dimension is often called an axis). The number of axes of a tensor is also called its rank.

Scalars (0D Tensors)

A tensor that contains only one number is called a scalar (or scalar tensor, or 0-dimensional tensor, or 0D tensor).

Here's a Numpy scalar:

```
import numpy as np

x = np.array(12)

print(x) # prints array(12)

print(x.ndim) # prints 0
```

Vectors (1D Tensors)

An array of numbers is called a vector, or 1D tensor. A 1D tensor is said to have exactly one axis. Following is a Numpy vector:

```
x = np.array([12, 3, 6, 14])

print(x) # => array([12, 3, 6, 14])

print(x.ndim) # => 1
```

Matrices (2D Tensors)

An array of vectors is a matrix, or 2D tensor. A matrix has two axes (often referred to rows and columns). You can visually interpret a matrix as a rectangular grid of numbers. This is a Numpy matrix:

```
x = np.array([[5, 78, 2, 34, 0],
              [6, 79, 3, 35, 1],
              [7, 80, 4, 36, 2]])
print(x.ndim ) # => 2
```

3d Tensors And Higher-dimensional Tensors

If you pack such matrices in a new array, you obtain a 3D tensor, which you can visually interpret as a cube of numbers. Following is a Numpy 3D tensor:

```
x = np.array([[[5, 78, 2, 34, 0],
               [6, 79, 3, 35, 1],
               [7, 80, 4, 36, 2]],
               [[5, 78, 2, 34, 0],
               [6, 79, 3, 35, 1],
               [7, 80, 4, 36, 2]],
               [[5, 78, 2, 34, 0],
               [6, 79, 3, 35, 1],
               [7, 80, 4, 36, 2]]])
print(x.ndim) # => 3
```

Key Attributes Of A Tensor

A tensor is defined by three key ：

1. Number of axes (rank)
2. Shape
3. Data type

Number Of Axes (Rank)

This is also called the tensor’s ndim in Python libraries such as Numpy. For instance, a 3D tensor has three axes, and a matrix has two axes .

Shape

This is a tuple of integers that describes how many dimensions the tensor has along each axis.

Data Type

This is the type of the data contained in the tensor; for instance, a tensor’s type could be float32, uint8, float64, and so on.

Manipulating Tensors In Numpy

All the operations we studied during numpy sessions, we can also perform them on tensors, e.g. slicing, indexing, fancy indexing, reshape, dot (Matrix multiplication), transpose, etc, etc

The Notion Of Data Batches

Deep-learning models don’t process an entire dataset at once; rather, they break the data into small batches and apply those batches to the neural network incrementally. Once a batch has passed the network and network has adjusted its parameters according to this batch a new batch of next n samples is extracted from dataset and applied over network.

Real-world Examples Of Data Tensors

Let's make data tensors more concrete with a few examples similar to what you'll encounter later. The data you'll manipulate will almost always fall into one of the following categories:

Vector data: 2D tensors of shape (samples, features)

Time Series data or sequence data: 3D tensors of shape (samples, timesteps, features)

Images: 4D tensors of shape (samples, height, width, channels) or (samples, channels, height, width)

Video: 5D tensors of shape (samples, frames, height, width, channels) or (samples, frames, channels, height, width)

Vector data

This is the most common case. In such a dataset, each single data point can be encoded as a vector, and thus a batch of data will be encoded as a 2D tensor (that is, an array of vectors), where the first axis is the samples axis and the second axis is the features axis. For example:

- An actuarial dataset of people, where we consider each person's age, ZIP code, and income. Each person can be characterized as a vector of 3 values, and thus an entire dataset of 100,000 people can be stored in a 2D tensor of shape (100000, 3).

Time Series data or sequence data

Whenever time matters in your data (or the notion of sequence order), it makes sense to store it in a 3D tensor with an explicit time axis. Each sample can be encoded as a sequence of vectors (a 2D tensor), and thus a batch of data will be encoded as a 3D tensor.

Image data

Images typically have three dimensions: height, width, and color depth. Although grayscale images (like our MNIST digits) have only a single color channel and could thus be stored in 2D tensors, by convention image tensors are always 3D. There are two conventions for shapes of images tensors: the channels-last convention (used by TensorFlow) and the channels-first convention (used by Theano). The TensorFlow machine-learning framework, from Google, places the color-depth axis at the end: (samples, height, width, color_depth). Meanwhile, Theano places the color depth axis right after the batch axis: (samples, color_depth, height, width)

Video data

Video data is one of the few types of real-world data for which you'll need 5D tensors. A video can be understood as a sequence of frames, each frame being a color image. Because each frame can be stored in a 3D tensor (height, width, color_depth), a sequence of frames can be stored in a 4D tensor (frames, height, width, color_depth), and thus a batch of different videos can be stored in a 5D tensor of shape (samples, frames, height, width, color_depth).

The Gears of Neural Networks

Tensor operations

Much as any computer program can be ultimately reduced to a small set of binary operations on binary inputs (AND, OR, NOR, and so on), all transformations learned by deep neural networks can be reduced to a handful of tensor operations applied to tensors of numeric data. For instance, it’s possible to add tensors, multiply tensors, and so on.

Element-wise operations

operations that are applied independently to each entry in the tensors being considered are called Element-wise operations.

Broadcasting

What happens with addition when the shapes of the two tensors being added differ?

When possible, and if there's no ambiguity, the smaller tensor will be broadcasted to match the shape of the larger tensor. Broadcasting consists of two steps:

1. Axes (called broadcast axes) are added to the smaller tensor to match the ndim of the larger tensor.
2. The smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor.

Tensor dot

The dot operation, also called a tensor product (not to be confused with an elementwise product) is the most common, most useful tensor operation. Contrary to element-wise operations, it combines entries in the input tensors.

Tensor reshaping

A third type of tensor operation that's essential to understand is tensor reshaping. Reshaping a tensor means rearranging its rows and columns to match a target shape. Naturally, the reshaped tensor has the same total number of coefficients as the initial tensor.

```
x = np.array([[0., 1.], [2., 3.], [4., 5.]])

print(x.shape)  => (3, 2)

x = x.reshape((6, 1))

print(x)  => [[ 0.], [ 1.], [ 2.], [ 3.], [ 4.], [ 5.]]
```

Geometric interpretation of tensor operations

Because the contents of the tensors manipulated by tensor operations can be interpreted as coordinates of points in some geometric space, all tensor operations have a geometric interpretation

A geometric interpretation of deep learning

You just learned that neural networks consist entirely of chains of tensor operations and that all of these tensor operations are just geometric transformations of the input data. It follows that you can interpret a neural network as a very complex geometric transformation in a high-dimensional space, implemented via a long series of simple steps.

The Engine of Neural Networks:

Gradient-Based Optimization

As we have seen in the previous section, each neural layer from our first network example transforms its input data as follows:

```
output = relu(dot(W, input) + b)
```

In this expression, W and b are tensors that are attributes of the layer. Initially, these weight matrices are filled with small random values (a step called random initialization). What comes next is to gradually adjust these weights, based on a feedback signal. This gradual adjustment, also called training, is basically the learning that machine learning is all about.

Gradient-Based Optimization

This happens within what's called a training loop, which works as follows. Repeat these steps in a loop, as long as necessary:

1. Draw a batch of training samples x and corresponding targets y.
2. Run the network on x (a step called the forward pass) to obtain predictions y_pred.
3. Compute the loss of the network on the batch, a measure of the mismatch between y_pred and y.
4. Update all weights of the network in a way that slightly reduces the loss on this batch.

Gradient-Based Optimization

A much better approach is to take advantage of the fact that all operations used in the network are differentiable, and compute the gradient of the loss with regard to the network's coefficients. You can then move the coefficients in the opposite direction from the gradient, thus decreasing the loss.

What's a derivative?

The **slope** a is called the derivative of $f(x)$. If a is negative, it means a small change of x around a point will result in a decrease of $f(x)$ (as shown in figure); and if a is positive, a small change in x will result in an increase of $f(x)$

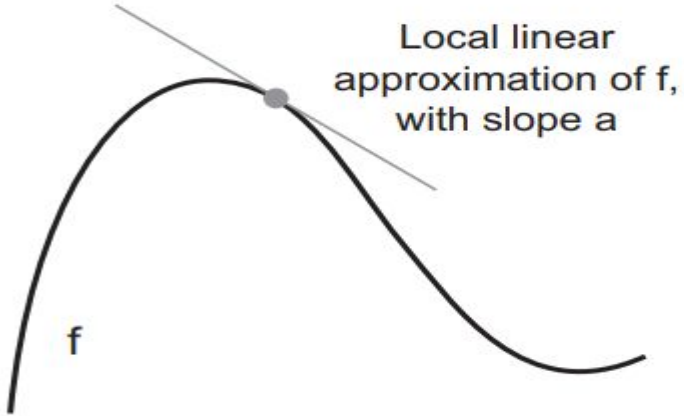


Figure 2.10

Derivative of a Tensor Operation

A gradient is the derivative of a tensor operation. It's the generalization of the concept of derivatives to functions of multidimensional inputs: that is, to functions that take tensors as inputs.

Stochastic gradient descent

Given a differentiable function, it's theoretically possible to find its minimum analytically: it's known that a function's minimum is a point where the derivative is 0, so all you have to do is find all the points where the derivative goes to 0 and check for which of these points the function has the lowest value. Applied to a neural network, that means finding analytically the combination of weight values that yields the smallest possible loss function. This is a polynomial equation of N variables, where N is the number of coefficients in the network. This is intractable for real neural networks, where the number of parameters is never less than a few thousand and can often be several tens of millions.

Stochastic gradient descent

Instead, you can use the four-step algorithm: modify the parameters little by little based on the current loss value on a random batch of data. Because you're dealing with a differentiable function, you can compute its gradient, which gives you an efficient way to implement step 4. If you update the weights in the opposite direction from the gradient, the loss will be a little less every time:

Mini-batch Stochastic Gradient Descent

The term stochastic refers to the fact that each batch of data is drawn at random (stochastic is a scientific synonym of random)

1. Draw a batch of training samples x and corresponding targets y.
2. Run the network on x to obtain predictions y_pred.
3. Compute the loss of the network on the batch, a measure of the mismatch between y_pred and y.
4. Compute the gradient of the loss with regard to the network's parameters (a backward pass).
5. Move the parameters a little in the opposite direction from the gradient for example W -= step * gradient thus reducing the loss on the batch a bit.

Chaining Derivatives: Backpropagation

Backpropagation algorithms are a family of methods used to efficiently train artificial neural networks (ANNs) following a gradient-based optimization algorithm that exploits the chain rule. The main feature of backpropagation is its iterative, recursive and efficient method for calculating the weights updates to improve the network until it is able to perform the task for which it is being trained.

Looking Back At Our First Example

```
model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
tf.keras.layers.Dense(512, activation=tf.nn.relu),
tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

Now we understand that this network consists of a chain of two Dense layers, that each layer applies a few simple tensor operations to the input data, and that these operations involve weight tensors. Weight tensors, which are attributes of the layers, are where the knowledge of the network persists.

Looking Back At Our First Example

```
model.compile(optimizer = tf.keras.optimizers.Adam(), loss =
'sparse_categorical_crossentropy', metrics = ['accuracy'])
```

This was the network-compilation step: Now we understand that the loss function is `sparse_categorical_crossentropy` that's used as a feedback signal for learning the weight tensors, and which the training phase will attempt to minimize. We also know that this reduction of the loss happens via minibatch stochastic gradient descent. The exact rules governing a specific use of gradient descent are defined by the `Adam` optimizer passed as the first argument.

Looking Back At Our First Example

Finally, this was the training loop:

```
model.fit(training_images, training_labels, epochs=5, batch_size=128)
```

Now we understand what happens when you call fit: the network will start to iterate on the training data in mini-batches of 128 samples, 5 times over (each iteration over all the training data is called an epoch). At each iteration, the network will compute the gradients of the weights with regard to the loss on the batch, and update the weights accordingly. At this point, we know most of what there is to know about neural networks.

Getting Started With Neural Networks

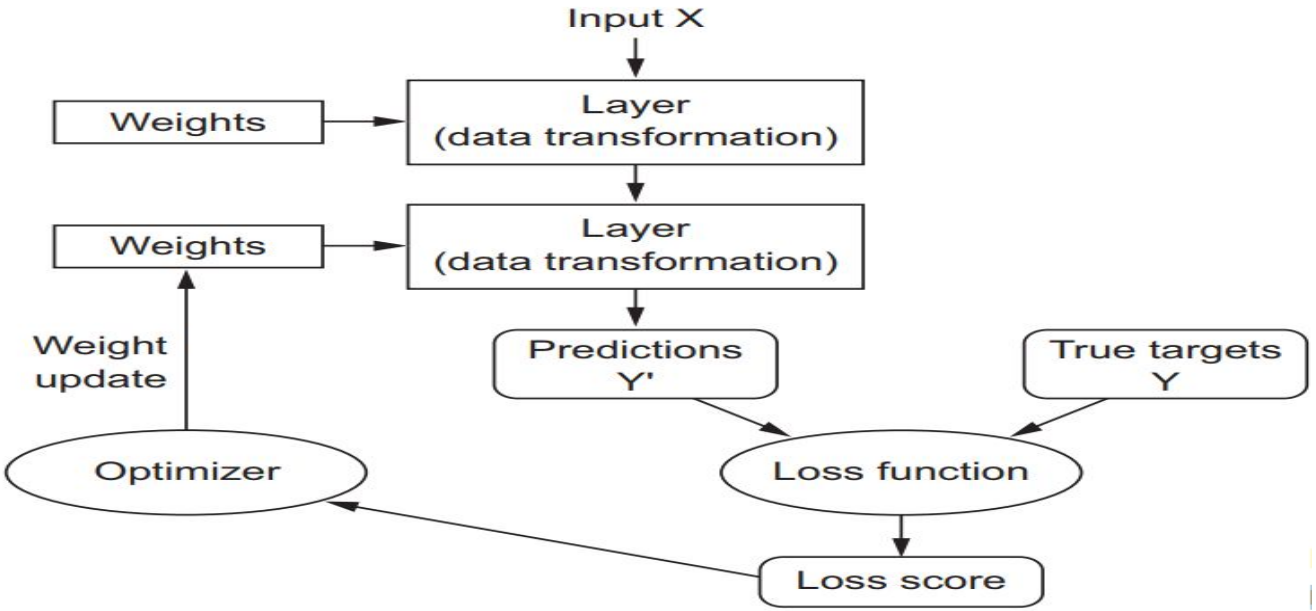
Chapter 3

Anatomy of a neural network

A neural network comprises of the following objects mainly:

- ***Layers***, which are combined to form a network (or model)
- ***Input data*** and corresponding targets
- ***Loss function***, which defines the feedback signal used for learning
- ***Optimizer***, which determines how learning proceeds

Network, Layers, Loss function, Optimizer Relation



Layers: The building blocks of deep learning

The fundamental data structure in neural networks is the layer. A layer takes as input one or more tensors and outputs one or more tensors. Usually layers store the state or knowledge in form of ***weights.*** There are different types of layers available for different tasks, like:

Dense Layers or ***Fully Connected*** Layers are used for 2D tensors of shape (samples, features)

Recurrent layers are used for sequence or 3D tensors of shape (samples, timesteps, features)

Layers: The building blocks of deep learning

2D convolution layers (Conv2D) are used for Image data, stored in 4D tensors

The notion of ***layer compatibility*** refers to the fact that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape. Consider the following example:

```
layer = layers.Dense(32, input_shape=(784,))
```

This layer accepts 784 features (axis 0) and unspecified / any number of samples (axis 1). The output of the layer is 32 at axis 1.

Layers: The building blocks of deep learning

In Keras, Layer object has built in feature to adopt to the shape of its input data. So the developer doesn't have to worry about it. In example we discussed previously,

```
model =
tf.keras.models.Sequential([tf.keras.layers.Flatten(),
tf.keras.layers.Dense(128, activation=tf.nn.relu),
tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

Here we can observe that every layer defines what it will output but not what it takes as input.

Models: networks of layers

A deep-learning model is a directed, acyclic graph of layers. The most common instance is a linear stack of layers, mapping a single input to a single output. But as you move forward, you'll be exposed to a much broader variety of network topologies. Some common ones include the following:

- Two-branch networks
- Multihead networks
- Inception blocks

Picking the right network architecture is more an art than a science; and although there are some best practices and principles you can rely on, only practice can help you become a proper neural-network architect

Loss functions and optimizers:

Once the network architecture is defined, you still have to choose two more things:

1. ***Loss function (objective function)***—The quantity that will be minimized during training. It represents a measure of success for the task at hand.
2. ***Optimizer***—Determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent (SGD).

Choosing the right objective function for the right problem is extremely important: your network will take any shortcut it can, to minimize the loss; so if the objective doesn't fully correlate with success for the task at hand, your network will end up doing things you may not have wanted

Introduction to Keras

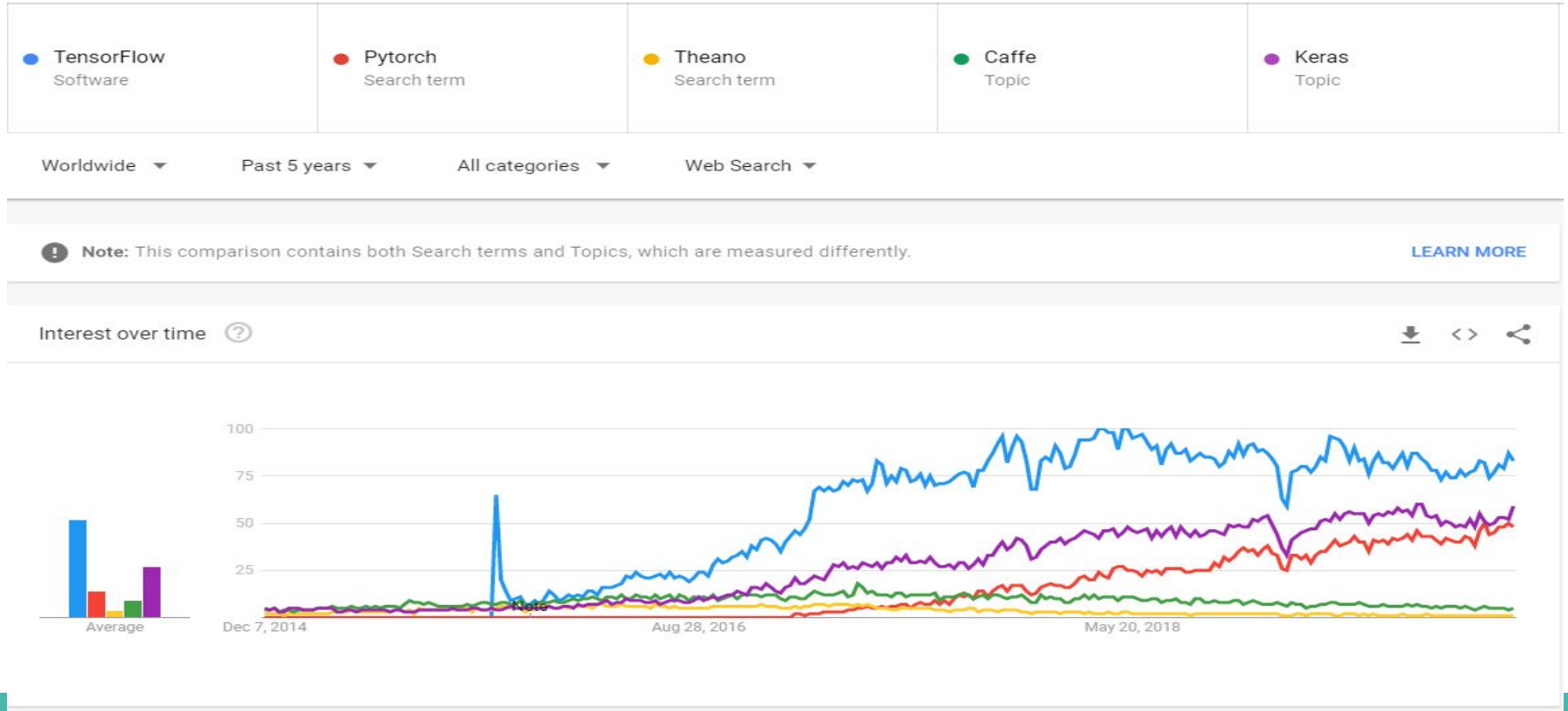
Keras is a deep-learning framework for Python that provides a convenient way to define and train almost any kind of deep-learning model. Keras was initially developed for researchers, with the aim of enabling fast experimentation. Keras has the following key features:

1. It allows the same code to run seamlessly on CPU or GPU.
2. It has a user-friendly API that makes it easy to quickly prototype deep-learning models.
3. It has built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both.

Introduction to Keras

4. It supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, and so on. This means Keras is appropriate for building essentially any deep-learning model, from a generative adversarial network to a neural Turing machine.

Google trends for deep-learning frameworks



Keras, TensorFlow, Theano, and CNTK

Keras is a model-level library, providing high-level building blocks for developing deep-learning models. Low level implementation and processing is done by backend engines Keras takes care model architecture at abstraction layer.

Currently Keras uses tensorflow as its default backend and has tight integration with tf.keras module in TensorFlow 2.0.

Via TensorFlow (or Theano, or CNTK), Keras is able to run seamlessly on both CPUs and GPUs. When running on CPU, TensorFlow is itself wrapping a low-level library for tensor operations called Eigen (http://eigen.tuxfamily.org)

.

Keras, TensorFlow, Theano, and CNTK

On GPU, TensorFlow wraps a library of well-optimized deep-learning operations called the NVIDIA CUDA Deep Neural Network library (cuDNN)

Developing with Keras: a quick overview

We've already seen one example of a Keras model: the Fashion MNIST example. The typical Keras workflow looks just like that example:

1. Define your training data: input tensors and target tensors.
2. Define a network of layers (or model) that maps your inputs to your targets.
3. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
4. Iterate on your training data by calling the fit() method of your model.

Developing with Keras: a quick overview

In keras there are two ways to create a model: one with instantiating Sequential class, we already have seen and practiced this. While the other is keras functional API way, which allows us to create arbitrary shaped networks for advanced use cases.

```
inputs = keras.Input(shape=(784,), name='img')
x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = keras.Model(inputs=inputs, outputs=outputs,
name='mnist_model')
```

Developing with Keras: a quick overview

```
(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
x_train = x_train.reshape(60000, 784).astype('float32') / 255
x_test = x_test.reshape(10000, 784).astype('float32') / 255

model.compile(loss='sparse_categorical_crossentropy',
              optimizer=keras.optimizers.RMSprop(), metrics=['accuracy'])

history = model.fit(x_train, y_train,
                    batch_size=64, epochs=5, validation_split=0.2)

test_scores = model.evaluate(x_test, y_test, verbose=2)
print('Test loss:', test_scores[0])
print('Test accuracy:', test_scores[1])
```

Setting up a deep-learning workstation (Demo)

Jupyter notebooks and Google Colab (Demo)

Running over CPU Vs GPU, Local Vs Cloud

Classifying movie reviews

A binary classification example

In this example, We'll learn to classify movie reviews as positive or negative, based on the text content of the reviews.

Classifying movie reviews (import modules)

```
import tensorflow as tf

from tensorflow.keras.datasets import imdb

from tensorflow.keras import models, layers, optimizers

import numpy as np

import matplotlib.pyplot as plt
```

Classifying movie reviews (Loading Dataset)

```
(train_data, train_labels), (test_data, test_labels) =
imdb.load_data(num_words=10000)

print(train_data[0])

print(train_labels[0])

#If training data is from Movie reviews it must have been in text, how come it is
numbers. Argument num_words restricts function to load only 10,000 most used
words.
```

Contd. (Loading Dataset)

```
max([max(sequence) for sequence in train_data])

# As Argument num_words restricts function to load only 10,000 most used words.
To check we can execute the above snippet it will display the max word index in
train_data
```

Contd. (Preparing the data)

```
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1
    return results
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Contd. (Building the network)

```
model = models.Sequential()

model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))

model.add(layers.Dense(16, activation='relu'))

model.add(layers.Dense(1, activation='sigmoid'))
```

Contd. (Building the network)

Because we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), it's best to use the binary_crossentropy loss. It isn't the only viable choice: we could use, for instance, mean_squared_error . But crossentropy is usually the best choice when we're dealing with models that output probabilities. Crossentropy is a quantity from the field of Information Theory that measures the distance between probability distributions or, in this case, between the ground-truth distribution and our predictions. In next slide is the step where we configure the model with the rmsprop optimizer and the binary_crossentropy loss function. Note that we'll also monitor accuracy during training.

Contd. (Compile the network)

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])

history = model.fit(partial_x_train, partial_y_train, epochs=20, batch_size=512,
validation_data=(x_val, y_val))
```

Contd. (Plotting the training and validation loss)

```
history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
plt.clf()
```

Contd. (Plotting the training and validation loss)

```
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']
plt.plot(epochs, acc_values, 'bo', label='Training acc')
plt.plot(epochs, val_acc_values, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Contd. (Predictions on new data)

```
model.predict(x_test)

# array([[0.00579366],
       [1.        ],
       [0.4512035 ],
       ...,
       [0.00185409],
       [0.00611657],
       [0.44180435]], dtype=float32)
```

Classifying newswires

A multiclass classification example

In this section, we'll build a network to classify Reuters newswires into 46 mutually exclusive topics. Because we have many classes, this problem is an instance of multi-class classification.

Single-label, multiclass classification

If each data point should be classified into one and only one category, the problem is more specifically an instance of single-label, multiclass classification.

Multilabel, multiclass classification

If each data point could belong to multiple categories (in this case, topics), this case is generally known as multilabel, multiclass classification problem.

The Reuters dataset

We'll work with the Reuters dataset, a set of short newswires and their topics, published by Reuters in 1986. It's a simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set. Like IMDB and MNIST , the Reuters dataset comes packaged as part of Keras

Classifying newswires (import modules)

```
import tensorflow as tf

from tensorflow.keras.datasets import reuters

from tensorflow.keras import models, layers, optimizers, utils

import numpy as np

import matplotlib.pyplot as plt
```

Classifying newswires (Loading Dataset)

```
(train_data, train_labels), (test_data, test_labels) =
reuters.load_data(num_words=10000)

print(train_data[0])

print(train_labels[0])
```

Classifying newswires (Loading Dataset)

As with the IMDB dataset, the argument num_words=10000 restricts the data to the 10,000 most frequently occurring words found in the data. We have 8,982 training examples and 2,246 test examples:

```
print(len(train_data)) # 8982
```

```
print(len(test_data)) # 2246
```

Classifying newswires (Preparing the data)

```
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Classifying newswires (Preparing the data)

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results
one_hot_train_labels = to_one_hot(train_labels)
one_hot_test_labels = to_one_hot(test_labels)
```

Classifying newswires (Preparing the data)

Instead of the approach in last slide we can create one hot encoded labels with these simple snippets.

```
one_hot_train_labels = utils.to_categorical(train_labels)
```

```
one_hot_test_labels = utils.to_categorical(test_labels)
```

Classifying newswires (Preparing Validation set)

Instead of the approach in last slide we can create one hot encoded labels with these simple snippets.

```
x_val = x_train[:1000]

partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]

partial_y_train = one_hot_train_labels[1000:]
```

Classifying newswires (Building the network)

```
model = models.Sequential()

model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(46, activation='softmax'))
```

Observe the difference from the network we created for imdb network, at the output layer we have softmax activation function instead of sigmoid other than layer size.

Classifying newswires (Building the network)

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['acc'])

history = model.fit(partial_x_train, partial_y_train, epochs=20, batch_size=512,

validation_data=(x_val, y_val))
```

Contd. (Plotting the training and validation loss)

```
history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
plt.clf()
```

Contd. (Plotting the training and validation loss)

```
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']
plt.plot(epochs, acc_values, 'bo', label='Training acc')
plt.plot(epochs, val_acc_values, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Contd. (Predictions on new data)

```
predictions = model.predict(x_test)

Predictions.shape # (2246, 46)

predictions[1].sum() # 1.000000

predictions[1].argmax() # 14
```

Contd. (using integer labels not one hot encoding)

Another way to encode the labels would be to cast them as an integer tensor, like this:

```
y_train = np.array(train_labels)

y_test = np.array(test_labels)
```

With integer labels, we should use sparse_categorical_crossentropy instead of categorical_crossentropy

```
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy',
metrics=['acc'])
```

Importance of Layer size

As there are 46 possible classes we should have intermediate layers greater or equal to 46. If we would have any intermediate layer size lesser than 46 will result in drop of accuracy.

Predicting house prices: Regression example

K Fold Cross Validation

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample.

The general procedure is as follows:

1. Shuffle the dataset randomly.
2. Split the dataset into k groups
3. For each unique group:
    1. Take the group as a hold out or test data set
    2. Take the remaining groups as a training data set
    3. Fit a model on the training set and evaluate it on the test set
    4. Retain the evaluation score and discard the model
4. Summarize the skill of the model using the sample of model evaluation scores

Regression example (import modules)

```
import tensorflow as tf

from tensorflow.keras.datasets import boston_housing

from tensorflow.keras import models, layers, optimizers, utils

import numpy as np

import matplotlib.pyplot as plt
```

Regression example (Loading Dataset)

```
(train_data, train_targets), (test_data, test_targets) =
boston_housing.load_data()

print(train_data.shape)

print(test_data.shape)

print(train_targets)
```

Regression example (Normalizing the data)

```
mean = train_data.mean(axis=0)

train_data -= mean

std = train_data.std(axis=0)

train_data /= std

test_data -= mean

test_data /= std
```

Regression example (Building the network)

```
def build_model():

  model = models.Sequential()

  model.add(layers.Dense(64, activation='relu',
input_shape=(train_data.shape[1],)))

  model.add(layers.Dense(64, activation='relu'))

  model.add(layers.Dense(1))

  model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

  return model
```

Regression example (K-Fold Validation)

```
k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
all_mae_histories = []
for i in range(k):
  print('processing fold #', i)
  val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
  val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]
  partial_train_data = np.concatenate([train_data[:i * num_val_samples],
train_data[(i + 1) * num_val_samples:]], axis=0)
  partial_train_targets = np.concatenate([train_targets[:i * num_val_samples],
train_targets[(i + 1) * num_val_samples:]], axis=0)
  model = build_model()
```

Regression example (K-Fold Validation)

```
  history = model.fit(partial_train_data, partial_train_targets,
epochs=num_epochs, batch_size=1, verbose=0)
  val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
  all_scores.append(val_mae)
  mae_history = history.history['mae']
  all_mae_histories.append(mae_history)
```

Regression example (ERROR)

```
average_mae_history = [ np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]

print('All fold Validation Errors: ', all_scores)

print('Mean of All fold Validation Errors: ', np.mean(all_scores))

print('average_mae_history: ', average_mae_history)
```

Regression example (Plotting ERROR)

```
plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)

plt.xlabel('Epochs')

plt.ylabel('Validation MAE')

plt.show()
```

Regression example (Final Model Training)

```
model = build_model()

model.fit(train_data, train_targets, epochs=80, batch_size=16, verbose=0)

test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)

print(test_mae_score)
```

Fundamentals of Machine Learning

Chapter 4

Four branches of machine learning

Supervised learning

This is by far the most common case. It consists of learning to map input data to known targets (also called labels), given a set of examples (often labeled by humans). All four examples we have encountered in this book so far are canonical examples of supervised learning.

Unsupervised learning

Unsupervised learning is a type of machine learning algorithm used to draw inferences from datasets consisting of input data without labeled responses.

The most common unsupervised learning method is cluster analysis, which is used for exploratory data analysis to find hidden patterns or grouping in data.

Self-supervised learning

Self-supervised learning (or self-supervision) is a relatively recent learning technique (in machine learning) where the training data is autonomously (or automatically) labelled. It is still supervised learning, but the datasets do not need to be manually labelled by a human, instead they are / can be labelled by finding and exploiting the relations (or correlations) between different inputs.

Reinforcement learning

In reinforcement learning, an agent receives information about its environment and learns to choose actions that will maximize some reward. For instance, a neural network that looks at a video game screen and outputs game actions in order to maximize its score can be trained via reinforcement learning.

Classification and regression glossary

Until now we have come across many terms involved in aggregation and classification, let's have a short recap of them:

Sample or input—Input of the Model

Prediction or output—Output of the Model

Target—The truth. What our model should ideally have predicted, according to an external source of data

Classification and regression glossary

Prediction error or loss value—A measure of the difference between our model's prediction and the target

Classes—A set of possible labels to choose from in a classification problem

Label—A specific instance of a class annotation in a classification problem.

Binary classification—A classification task where each input sample should be categorized into one of two exclusive categories

Classification and regression glossary

Multiclass classification—A classification task where each input sample should be categorized into one category from a choice of more than two categories

Multilabel classification—A classification task where each input sample can be assigned multiple labels

Scalar regression—A task where the target is a continuous scalar value. Predicting house prices is a good example

Vector regression—A task where the target is a set of continuous vector values

Classification and regression glossary

Mini-batch or batch—A small set of samples (typically between 8 and 128) that are processed simultaneously by the model. The number of samples is often a power of 2, to facilitate memory allocation on GPU. When training, a mini-batch is used to compute a single gradient-descent update applied to the weights of the model.

Evaluating machine-learning models

Generalization

Generalization usually refers to a ML model's ability to perform well on new unseen data rather than just the data that it was trained on. The goal of a good machine learning model is to generalize well from the training data to any data from the problem domain. This allows us to make predictions in the future on data the model has never seen.

Underfitting

Underfitting occurs when a model is too simple — informed by too few features or regularized too much — which makes it inflexible in learning from the dataset. Simple learners tend to have less variance in their predictions but more bias towards wrong outcomes

Overfitting

Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the models ability to generalize

Underfitting vs Overfitting vs Generalization

Underfit Model. A model that fails to sufficiently learn the problem and performs poorly on a training dataset and does not perform well on a holdout sample

Overfit Model. A model that learns the training dataset too well, performing well on the training dataset but does not perform well on a hold out sample

Good Fit Model. A model that suitably learns the training dataset and generalizes well to the old out dataset

Training, Validation and Testing

Evaluating a model generally boils down to splitting the available data into three sets: training, validation, and test. We train on the training data and evaluate our model on the validation data. Once our model is ready for prime time, we test it one final time on the test data. Training a model involves tuning its hyperparameters, based on the validation loss / error. Every time we tune the model to perform good on validation data, we leak some information about the data into model. Hence as we achieve the desired validation accuracy we evaluate the model on test dataset, which our model has not encountered during training and validation.

Splitting Techniques For Dataset

SIMPLE HOLD - OUT VALIDATION: As we have seen in previous slides in this technique we split dataset into three chunks, one for training, one for validation and last one for testing.

Splitting Techniques For Dataset

K- FOLD VALIDATION: With this approach, we split our data into K partitions of equal sizes. For each partition i , train the model on the remaining K – 1 partitions, and evaluate it on partition i . Our final score is then the average of the K scores obtained. This method is helpful when the performance of our model shows significant variance based on our train-test split. Like hold-out validation, this method doesn't exempt us from using a distinct validation set for model calibration / testing

Splitting Techniques For Dataset

ITERATED K- FOLD VALIDATION WITH SHUFFLING: This one is for situations in which we have relatively little data available and we need to evaluate our model as precisely as possible. This approach has been found extremely helpful in Kaggle competitions. It consists of applying K-fold validation multiple times, shuffling the data every time before splitting it K ways. The final score is the average of the scores obtained at each run of K-fold validation. Note that we end up training and evaluating P × K models (where P is the number of iterations we used), which can be very expensive.

Splitting Techniques (Things to keep in mind)

Data representativeness—Our training set and test set must be representative of the data at hand, they must contain examples of all the classes. For this reason, we must randomly shuffle our data before splitting it into training and test sets.

The arrow of time—If we're trying to predict the future given the past (for example, tomorrow's weather, stock movements, and so on), we should not shuffle our data before splitting it, because doing so will create a temporal leak: our model will effectively be trained on data from the future. In such situations, we should always make sure all data in our test set is posterior to the data in the training set.

Splitting Techniques (Things to keep in mind)

Redundancy in data—If some data points in our data appear twice (fairly common with real-world data), then shuffling the data and splitting it into a training set and a validation set will result in redundancy between the training and validation sets. In effect, we'll be testing on part of our training data, which is the worst thing anyone can do! Make sure our training and validation sets are disjoint.

Data preprocessing, feature engineering, and feature learning

Data preprocessing for neural networks

Data preprocessing aims at making the raw data at hand more amenable to neural networks. This includes **Vectorization**, **Normalization**, **Handling Missing values**, and **feature extraction**

Data preprocessing (Vectorization)

All inputs and targets in a neural network must be tensors of floating-point data (or, in specific cases, tensors of integers). Whatever data we need to process—sound, images, text; we must first turn into tensors, this step is called **data Vectorization.**

Data preprocessing (Normalization)

In general, it isn't safe to feed into a neural network data that has large values (for example, multidigit integers, which are much larger than the initial values taken by the weights of a network) or data that is heterogeneous (for example, data where one feature is in the range 0–1 and another is in the range 100–200). Doing so can trigger large increase in weights during updates that will prevent the network from converging.

Data preprocessing (Normalization)

To make learning easier for our network, our data should have the following characteristics:

Take small values—Typically, most values should be in the 0–1 range.

Be homogenous—That is, all features should take values in roughly the same range.

Data preprocessing (Normalization)

Additionally, the following stricter normalization practice is common and can help, although it isn't always necessary (for example, we didn't do this in the digit-classification example):

- Normalize each feature independently to have a mean of 0.
- Normalize each feature independently to have a standard deviation of 1.

This is easy to do with Numpy arrays:

```
x -= x.mean(axis=0)

x /= x.std(axis=0)
```

Data preprocessing (Handling Missing Values)

Data in real world are rarely clean and homogeneous. Typically, they tend to be incomplete, noisy, and inconsistent and it is an important task of a Data scientist to preprocess the data by dealing with missing values properly. Missing values could be: NaN, empty string, ?,-1,-99,-999 and so on. In general, with neural networks, it’s safe to input missing values as 0, with the condition that 0 isn’t already a meaningful value. The network will learn from exposure to the data that the value 0 means missing data and will start ignoring the value

Handling Missing Values (Common Techniques)

- Replace value, Backward fill, forward fill
- Replace value by mean, median or mode
- Drop records / samples having missing values
- Replace missing values using supervised learning, classification or regression

Data preprocessing (Feature engineering)

Feature engineering efforts mainly have two goals:

- Preparing the proper input dataset, compatible with the machine learning algorithm requirements.
- Improving the performance of machine learning models.

The features you use influence more than everything else the result. No algorithm alone, to my knowledge, can supplement the information gain given by correct feature engineering. — Luca Massaron (Data Scientist / Author / Google Developer Expert in Machine Learning)

Overfitting and Underfitting

Overfitting and Underfitting

The fundamental issue in machine learning is the balancing between *optimization* and *generalization*. *Optimization* refers to the process of adjusting a model to get the best performance possible on the training data (the learning in machine learning), whereas *generalization* refers to how well the trained model performs on data it has never seen before.

Overfitting and Underfitting

Underfitting refers to a model that can neither model the training data nor generalize to new data. An underfit machine learning model is not a suitable model and will be obvious as it will have poor performance on the training data.

Whereas ***Overfitting*** is a modeling error that occurs when a function is too closely fit to a limited set of data points

Overcome Underfitting

To cope with underfitting we can:

- Introduce more complexity in dataset like:
  - add more layers
  - increase layer sizes
- Add more features to dataset

Overcome Overfitting

To cope with overfitting we can:

- Get more training data
- Reducing the network’s size
- Add regularization
- Add dropout layers

Overcome Overfitting (Get more training data)

To prevent a model from learning misleading or irrelevant patterns found in the training data, the best solution is to get more training data. A model trained on more data will naturally generalize better.

Overcome Overfitting (Reducing the network's size)

In deep learning, the number of learnable parameters in a model is often referred to as the model's capacity. Intuitively, a model with more parameters has more memorization capacity and therefore can easily learn a perfect dictionary-like mapping between training samples and their targets a mapping without any generalization power. On the other hand, if the network has limited memorization resources, it won't be able to learn this mapping as easily; thus, in order to minimize its loss, it will have to resort to learning compressed representations that have predictive power regarding the targets precisely the type of representations we're interested in.

Demo on imdb Dataset

Overcome Overfitting (Weight regularization)

A simple model is a model where the distribution of parameter values has less entropy (or a model with fewer parameters, as we saw in the previous section). Thus a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to take only small values, which makes the distribution of weight values more regular, this is called weight regularization.

Overcome Overfitting (Weight regularization)

Weight regularization is done by adding to the loss function of the network a cost associated with having large weights. This cost comes in two flavors:

- **L1 regularization**: The cost added is proportional to the absolute value of the weight coefficients (the L1 norm of the weights).
- **L2 regularization**: The cost added is proportional to the square of the value of the weight coefficients (the L2 norm of the weights).

Overcome Overfitting (Weight regularization)

In Keras, weight regularization is added by passing weight regularizer instances to layers as keyword arguments, e.g

```
model = models.Sequential()

model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
activation='relu', input_shape=(10000,)))

model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
activation='relu'))

model.add(layers.Dense(1, activation='sigmoid'))
```

Overcome Overfitting (Weight regularization)

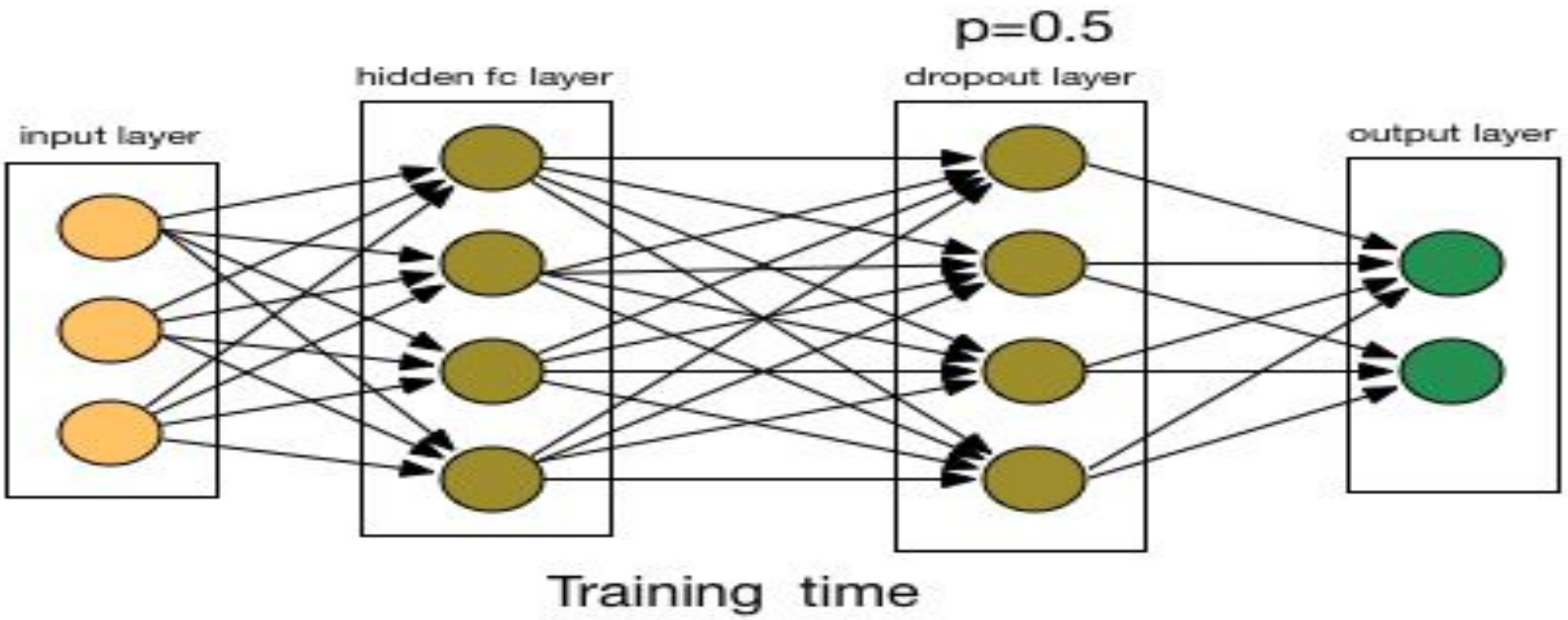
Always remember that the regularization penalty is applied at the time of training only, when the weights are learned during training.

Overcome Overfitting (Add dropout layers)

Dropout is one of the most effective and most commonly used regularization techniques for neural networks, developed by Geoff Hinton and his students at the University of Toronto. Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training. The dropout rate is the fraction of the features that are zeroed out; it's usually set between 0.2 and 0.5. At test time, no units are dropped out; instead, the layer's output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time. In keras we add dropout layer as we add other layers

```
model.add(layers.Dropout(0.5))
```

Overcome Overfitting (Add dropout layers)



The universal workflow of machine learning

The universal workflow of machine learning

Universal blueprint that we can use to attack and solve any machine-learning problem ties together the concepts we've learned about in this chapter:

problem definition, evaluation, feature engineering, and fighting overfitting.

Defining the problem and assembling a dataset

First, we must define the problem at hand:

- What will our input data be?
- What are we trying to predict?
- What type of problem are we facing?
- Is it binary classification?
- Multiclass classification?
- Scalar regression?
- Vector regression? Multiclass, multilabel classification? Something else, like clustering, generation, or reinforcement learning?

Defining the problem and assembling a dataset

We can’t move to the next stage until we know what our inputs and outputs are, and what data we’ll use. Following are the hypotheses we should make at this stage:

- We hypothesize that our outputs can be predicted given our inputs.
- We hypothesize that our available data is sufficiently informative to learn the relationship between inputs and outputs.

Choosing a measure of success

To control something, we need to be able to observe it. To achieve success, we must define what we mean by success accuracy? Precision and recall? Customer-retention rate? Our metric for success will guide the choice of a loss function: what our model will optimize. It should directly align with our higher-level goals, such as the success of our business.

Deciding on an evaluation protocol

Once we know what we're aiming for, we must establish how we'll measure our current progress. We've previously reviewed three common evaluation protocols:

- Maintaining a hold-out validation set: The way to go when you have plenty of data
- Doing K-fold cross-validation: The right choice when you have too few samples for hold-out validation to be reliable
- Doing iterated K-fold validation: For performing highly accurate model evaluation when little data is available

For evaluation Just picking one of these will work. In most cases, the first will work well enough.

Preparing our data

We should format our data in a way that can be fed into a machine-learning model here, we'll assume a deep neural network:

- As we saw previously, our data should be formatted as tensors.
- The values taken by these tensors should usually be scaled to small values
- If different features take values in different ranges (heterogeneous data), then the data should be normalized.
- We may want to do some feature engineering, especially for small-data problems.

Once our tensors of input data and target data are ready, we can begin to train model.

Developing a model that does better than a baseline

Before starting the model definition and training we must define a baseline for success criterion. If we can not create a model after many successive tries, means that the hypothesis we built at step 1 are false and we need to move back to step 1 for gathering input data which can predict the output.

Developing a model that does better than a baseline

Assuming that things go well, then we need to make three key choices to build our first working model:

- Last-layer activation: This establishes useful constraints on the network's output. For instance, the IMDB classification example used sigmoid in the last layer; the regression example didn't use any last-layer activation; and so on.
- Loss function: This should match the type of problem we're trying to solve. For instance, the IMDB example used binary_crossentropy, the regression example used mse, and so on.
- Optimization configuration: What optimizer will we use? What will its learning rate be? In most cases, it's safe to go with rmsprop and its default learning rate.

Developing a model that does better than a baseline

Following table can help us choose a last-layer activation and a loss function for a few common problem types.

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

Scaling up: developing a model that overfits

Once we have built a model that satisfies the baseline criterion, now we need to make it powerful enough that optimizes accuracy without compromising the generalization. Remember the Generalization lies between the underfitting and overfitting. The easy way to get to it by just reaching the border towards overfitting. Once our model starts overfitting we know how to cope up with overfitting.

To make model overfit:

- Add layers
- Make the layers bigger
- Train for more epochs

Regularizing model and tuning hyperparameters

This step will take the most time: we'll repeatedly modify our model, train it, evaluate on our validation data (not the test data, at this point), modify it again, and repeat, until the model is as good as it can get. These are some things to try:

- Add dropout.
- Try different architectures: add or remove layers.
- Add L1 and/or L2 regularization.
- Try different hyperparameters (such as the number of units per layer or the learning rate of the optimizer) to find the optimal configuration.
- Optionally, iterate on feature engineering: add new features, or remove features that don't seem to be informative.

Chapter 4 Ends Here

Part 2

Deep learning in practice

Chapters 5–9 will help you gain practical intuition about how to solve real-world problems using deep learning, and will familiarize you with essential deep-learning best practices. Most of the code examples in the book are concentrated in this second half.

5 Deep learning for computer vision

This chapter covers

- Understanding convolutional neural networks (convnets)

Intro to Convolutional Neural Networks-Convnets

Listing 5.1 Instantiating a small convnet

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Nuts and Bolts of a Convnet

Let explore the concept of convolutional Neural Network:

Convolutional Layers(stateful layer):

- Convolutional layer is used here as Com,nv2D
- Input shape is a 3D tensor with shape (28,28,1)
- Activation function “Relu” Rectified linear unit is used
- Output neurons are 32 dimensional
-
- There is a (3,3) kernel is used

Max Polling Laying Layers(stateless layer)

-A max polling 2D layer is used with channel size(2,2)

What is convolution?

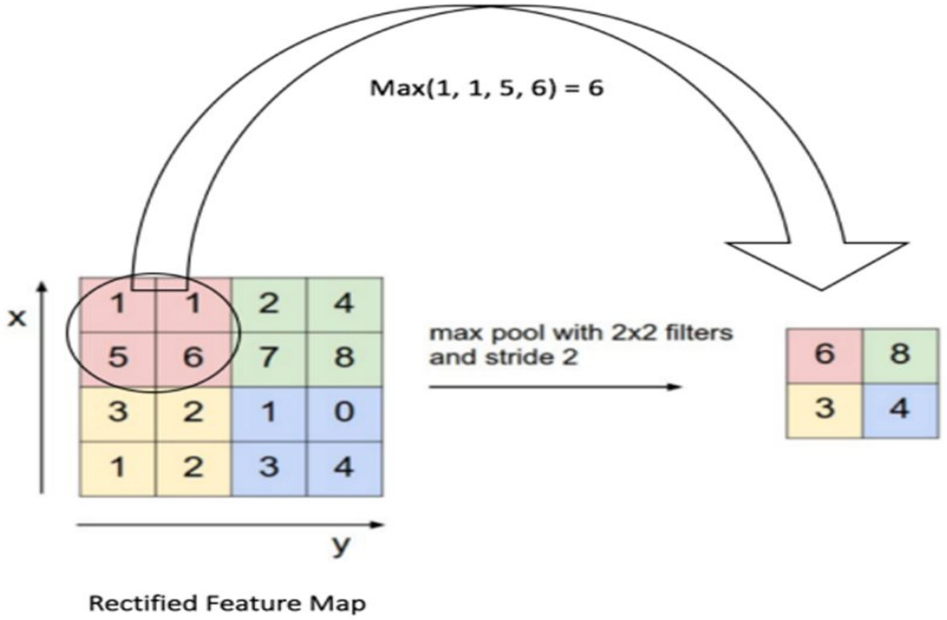
In mathematics convolution is a mathematical operation on two functions that produces a third function expressing how the shape of one is modified by the other. The term convolution refers to both the result function and to the process of computing it.

In our convolutional layers these two convolving functions are:

- Our input image (28,28,1)
- The kernal (3,3)

Max Pooling

Maximum pooling, or **max pooling**, is a **pooling** operation that calculates the **maximum**, or largest, value in each patch of each feature map. The results are down sampled or **pooled** feature maps that highlight the most present feature in the patch



Model Summary:

Summary is important to understand as it will actually clear the concept of why convolutional layer is preferred over dense layer for image processing

```
>>> model.summary()

Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D)  (None, 13, 13, 32)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D)  (None, 5, 5, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                36928
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```