FUNDAMENTALS OF DEEP LEARNING

Chapter 1

What will we cover

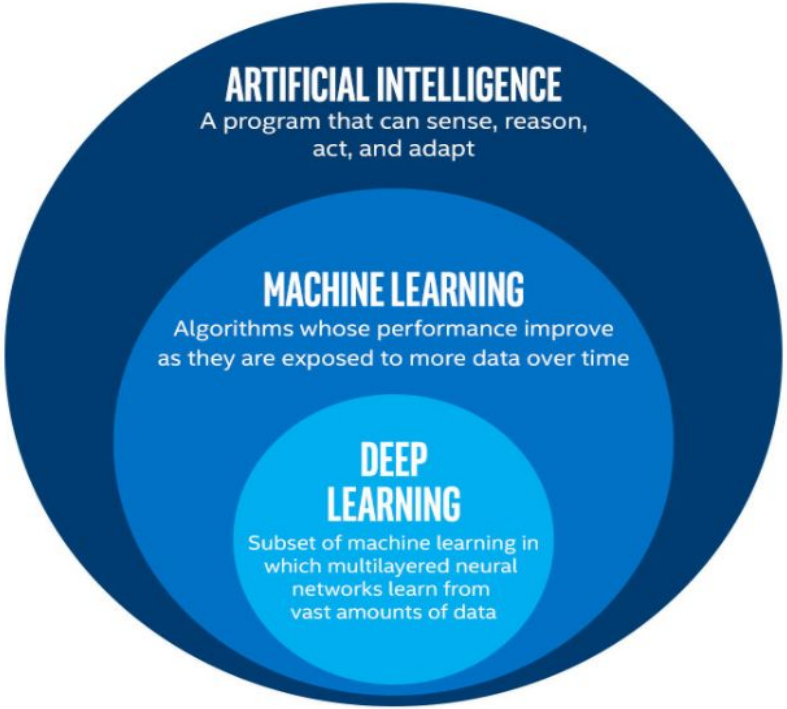
- What is Machine Learning
- Fundamental concepts involved in Machine Learning
- Four Branches of Machine Learning
- What is Deep Learning
- How it works
- What it can achieve

Text Book / Reference Book

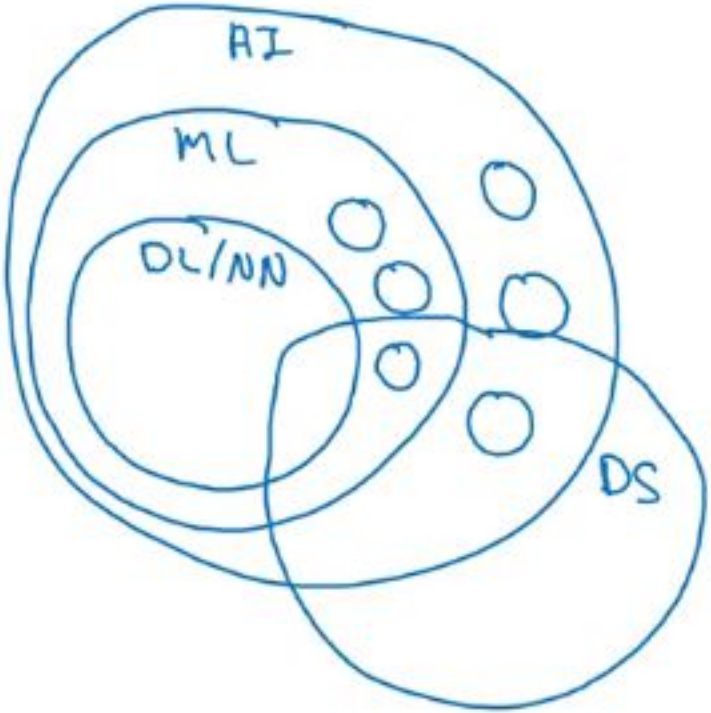
- Deep Learning with Python, Authored by FRANÇOIS CHOLLET

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.

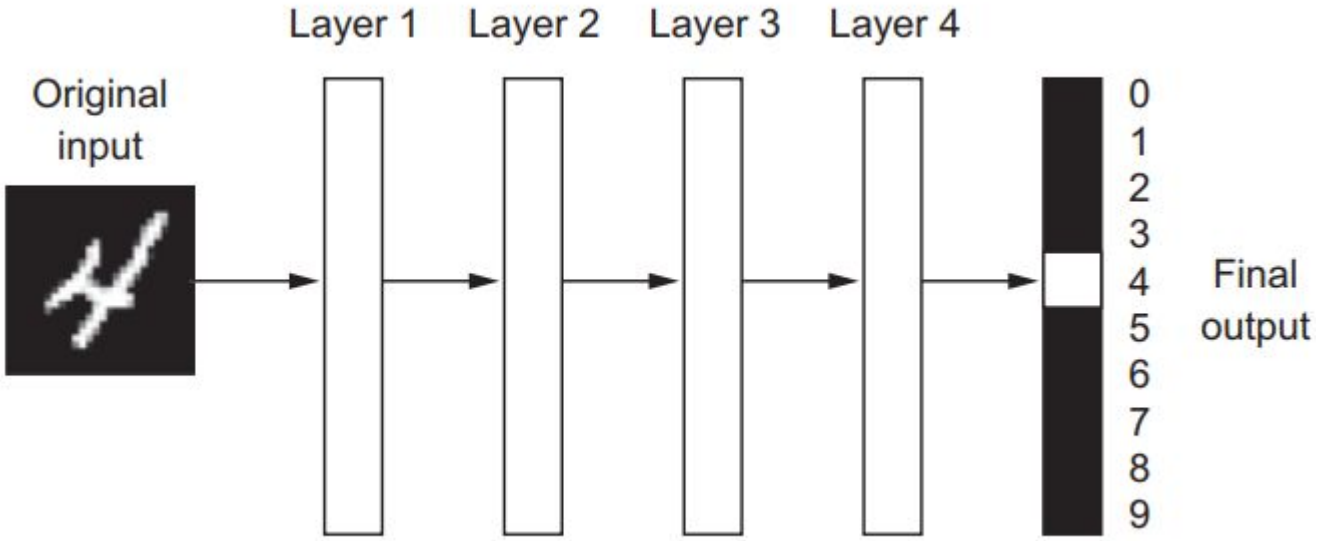
Reference:
https://www.prowesscorp.com/whats-the-difference-between-artificial-intelligence-ai-machine-learning-and-deep-learning/

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



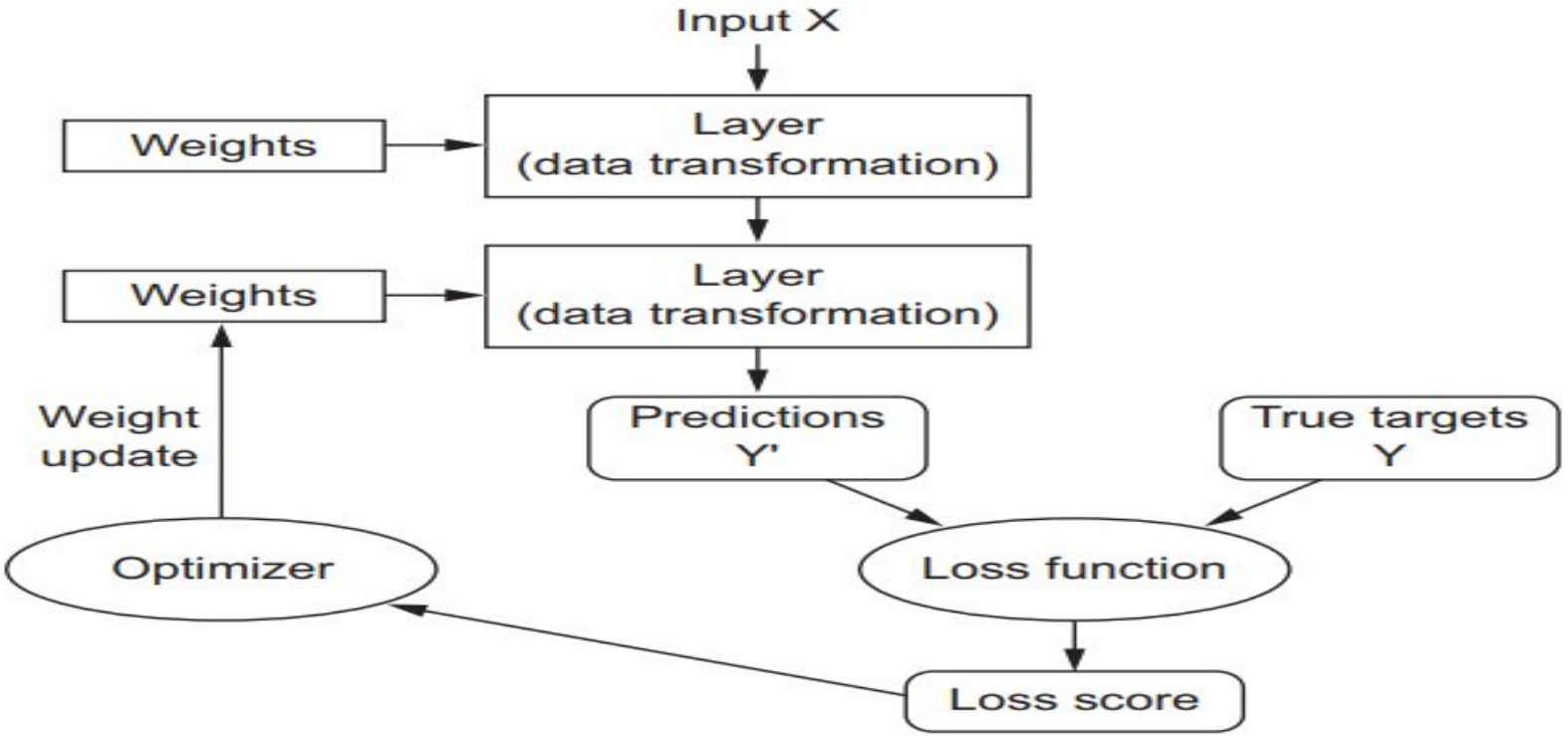
So that’s what deep learning is, technically:

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.

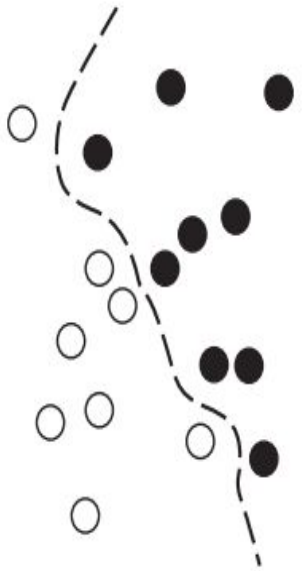
Early Neural Networks

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

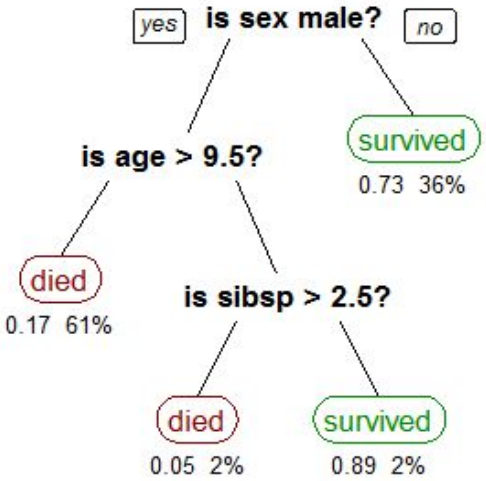
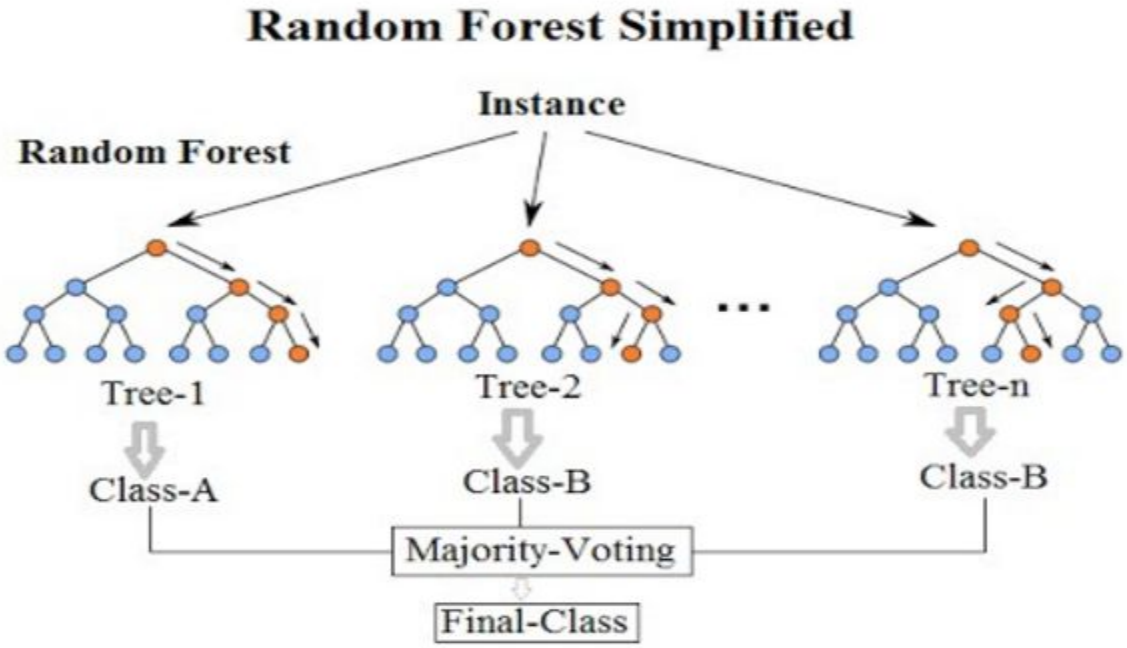


Image taken from wikipedia

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.

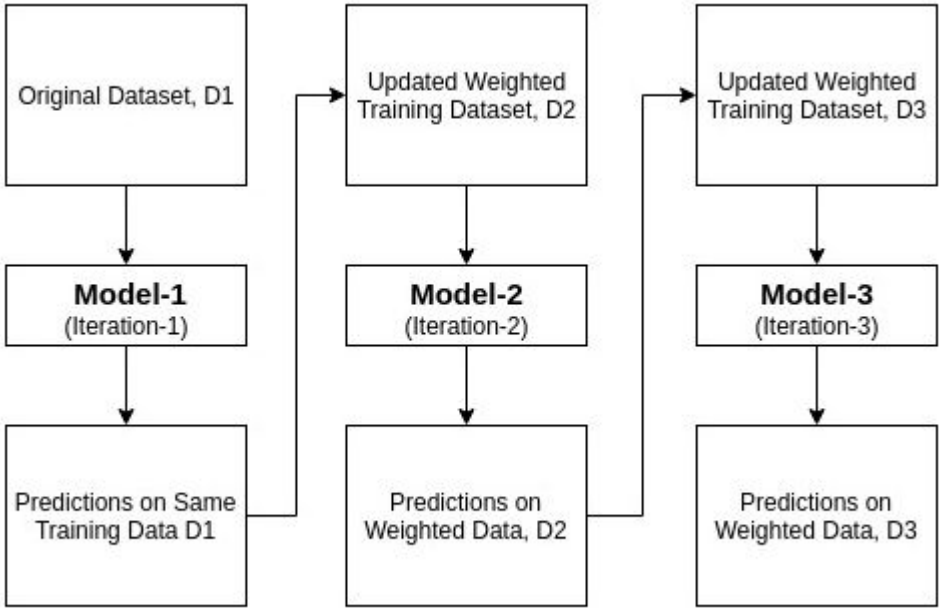
Random Forest Example



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 $f(x)$

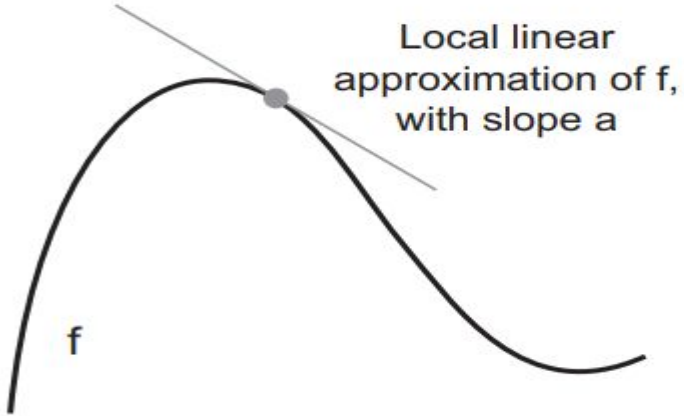


Figure 2.10

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

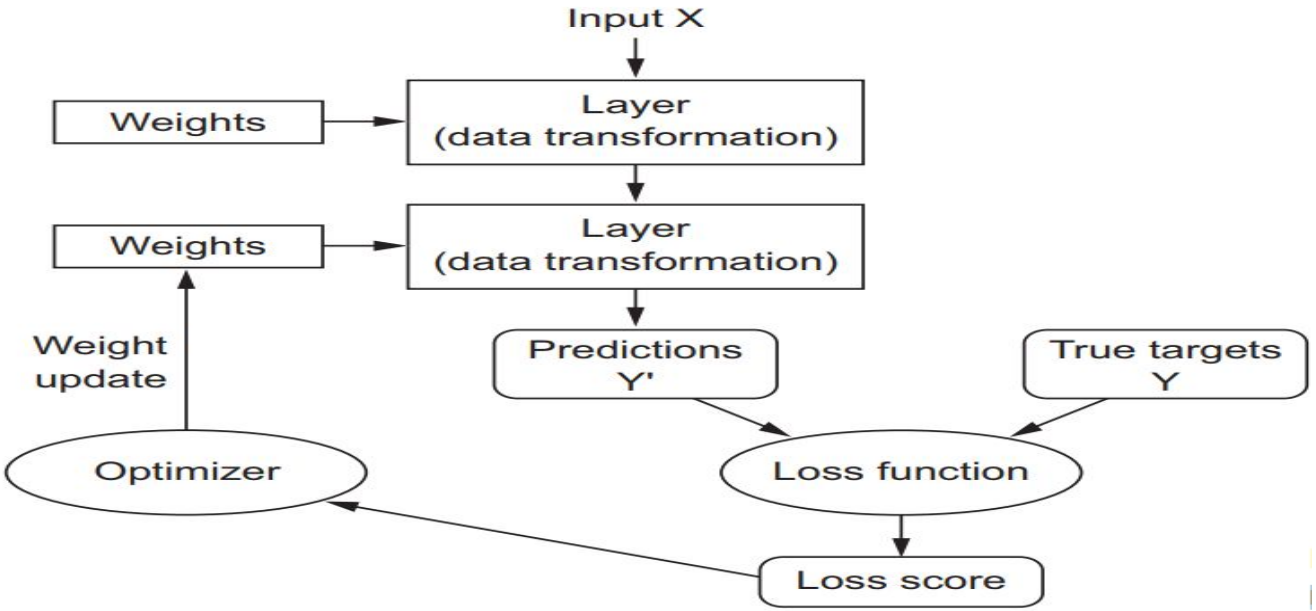
Chapter 3

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

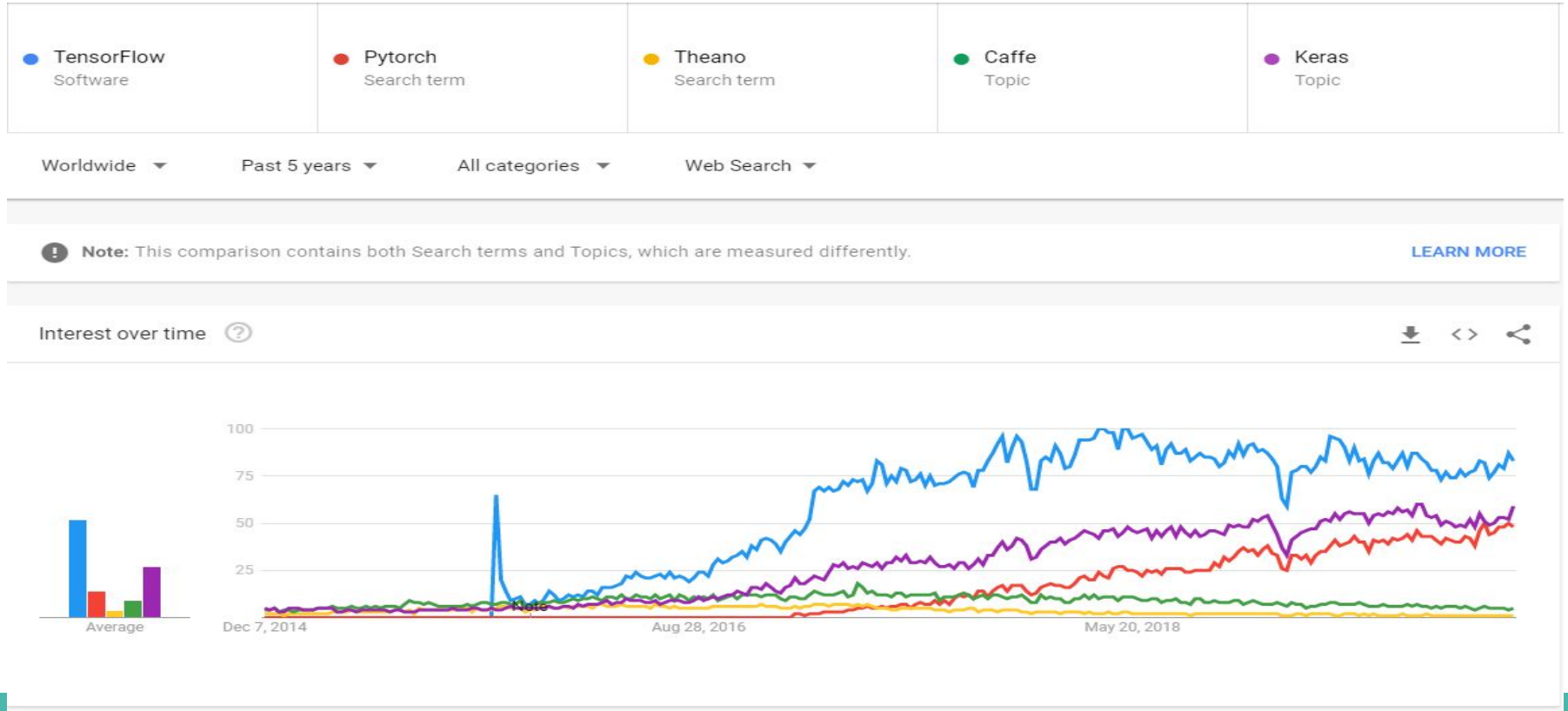
Keras is a deep-learning framework for Python that provides a convenient way to define and train almost any kind of deep-learning model. Keras was initially developed for researchers, with the aim of enabling fast experimentation. Keras has the following key features:

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