A Concise Handbook of TensorFlow

Release 0.3 beta

Xihan Li（雪麒）

Aug 28, 2018

Contents

基于 *Eager Execution* / *Based on Eager Execution*

本手册是一篇精简的 TensorFlow 入门指导，基于 TensorFlow 的 Eager Execution（动态图）模式，力图让具备一定机器学习及 Python 基础的开发者们快速上手 TensorFlow。

友情提醒：如果发现阅读中有难以理解的部分，请检查自己对每章的“前置知识”部分是否有清楚的理解。

答疑区 - TensorFlow 中文社区“简单粗暴 TensorFlow”版面：https://www.tensorflowers.cn/b/48 （如果您对本教程有任何疑问，请至 TensorFlow 中文社区的该版面发问）

PDF 下载：https://www.tensorflowers.cn/t/6230

GitHub: https://github.com/snowkylin/TensorFlow-cn

This handbook is a concise introduction to TensorFlow based on TensorFlow’s Eager Execution mode, trying to help developers get started with TensorFlow quickly with some basic machine learning and Python knowledge.

Friendly reminder: If you find something difficult to understand in reading, please check if you have a clear understanding of the “Prerequisites” part of each chapter.

Q&A area - TensorFlow Chinese community “A Concise Handbook of TensorFlow” forum: https://www.tensorflowers.cn/b/48 (If you have any questions about this tutorial, please ask in this forum of the TensorFlow Chinese community)

PDF download: https://www.tensorflowers.cn/t/6230

GitHub: https://github.com/snowkylin/TensorFlow-cn

CHAPTER 1

Preface

On Mar. 30th, 2018, Google held the second TensorFlow Dev Summit in Mountain View, California and announced the official release of TensorFlow version 1.8. I was fortunate to attend the summit with Google' s sponsorship, witnessing the release of this new version, a milestone. Lots of new functions added and supported shows the ambition of TensorFlow. Meanwhile, Eager Execution, which has been tested since 2017 fall, was finally included officially in this version and became the recommended mode for newcomers of TensorFlow.

The easiest way to get started with TensorFlow is using Eager Execution.

——https://www.tensorflow.org/get_started/

Before then, the disadvantages of Graph Execution mode in TensorFlow, such as high learning threshold, difficulty in debugging, poor flexibility and inability to use Python native controlling statements, have already been criticized by developers for a long time. Some new deep learning frameworks based on dynamic computational graph (e.g. PyTorch) have come out and won their places by their usability and efficiency for development. These dynamic deep learning frameworks are popular especially in academic researches where fast iterative development of models are required. In fact, I was the only person who used "old-fashioned" TensorFlow in my machine learning laboratory where I worked with dozens of colleagues. However, until now, most of the Chinese technical books and materials about TensorFlow still based on Graph Execution mode, which really dissuades beginners (especially those undergraduates who have just finished their machine learning courses) from learning. Therefore, as TensorFlow officially supports Eager Execution, it' s necessary to publish a brand new handbook which helps beginners and researchers who need to iterate models rapidly and to get started quickly from a new perspective.

Meanwhile, this handbook has another mission. Most Chinese technical books about TensorFlow focus mainly on deep learning and regard TensorFlow as a mere tool to implement deep learning models. Admittedly, they are self-contained, but it' s not friendly enough for those who have already known about machine learning and deep learning theories and want to focus on learning TensorFlow itself. In addition, though TensorFlow has its official documentation (https://tensorflow.google.cn/tutorials), its structure is not well organized, lacking the step-by-step feature of a common tutorial, thus being more similar to a technological documentation.

The main features of this handbook are:

- This book is mainly based on the most up-to-date Eager Execution mode in TensorFlow for fast iterative development of models. However, traditional Graph Execution mode is also included and we will do our best to make the codes provided in this book compatible with both modes.
- We position this book mainly as a tutorial and handbook, and arrange the concepts and functions of TensorFlow as the core part, for TensorFlow developers to refer quickly. Chapters are relatively independent with one another, therefore it's not necessary to read this book in a sequential order. There won't be much theory of deep learning and machine learning in the text, however some recommendation is still provided for beginners to grasp related basic knowledge.
- All codes are carefully written in order to be concise and efficient. All models are implemented based on the `tf.keras.Model` and `tf.keras.layers.Layer` methods, which just proposed by TensorFlow official documentation and are barely introduced in other technical documentations. This implementation guarantees high reusability. Each project is written by codes fewer than 100 lines for readers to understand and practice quickly.
- Less is more. No all-rounded or large blocks of details.

The parts marked "*" are optional in this handbook.

This handbook is tentatively named as "A Concise Handbook of TensorFlow" in order to pay a tribute to the book "A Concise Handbook of $\LaTeX$" written by my friend and colleague Chris Wu. The latter is a rare Chinese material about $\LaTeX$. I also learned from it while I was writing this handbook. This handbook was initially written and used by meself as a prerequisite handout in a deep learning seminar organized by my friend Ji-An Li. My friends' wise and selflessness also prompted me to finish this work.

The English version of this handbook is translated by my friend Zida Jin (Chapter 1-4) and Ming (Chapter 5-6), and revised by Ji-An Li and me. My three friends sacrificed a lot of valuable time to translate and proofread this handbook. Ji-An Li also provided valuable comments on the teaching content and code details of this manual. I would like to express my heartfelt thanks to my friends for their hard work.

I am grateful to the members of the Google China Developer Relations team and the TensorFlow engineering team for their help in writing this handbook. Among them, Luke Cheng of the Developer Relations team provided inspiration and continuous encouragement throughout the writing of this manual. Rui Li and Pryce Mu of the Developer Relations team provided strong support in the promotion of this manual, as well as Tiezhen Wang of TensorFlow team provided many suggestions to the engineering details of the manual.

Xihan Li (Snowkylin)

August 2018 in Yanyuan

CHAPTER 2

TensorFlow Installation

The most up-to-date installation method can be acquired through the official website (https://tensorflow.google.cn/install). TensorFlow supports multiple programming languages like Python, Java and Go and a variety of operating systems like Windows, OSX and Linux. We prefer Python in this handbook.

We provide both simple and full installation methods for readers with different requirements.

2.1 Simple Installation

You' d better follow these steps if you only want to install TensorFlow on your personal computer without GPU or spending too much effort configuring the environment (taking Windows as an example):

- Download and install Python IDE Anaconda (with Python Ver 3.6);
- Download and install Python IDE PyCharm (Community version, students can apply for free Professional versions)
- Run `Anaconda Prompt` in the Start Menu, enter and execute `pip install tensorflow`.

And done.

2.2 Full Installation

This part includes more details of installation (e.g. building a conda environment) and guidance for the GPU version of TensorFlow environment installation.

2.2.1 Environment configuration before installation

Before installing TensorFlow, we need to set up a proper environment with the following steps:

1. Check if your computer has an NVIDIA graphics card and install the GPU version of TensorFlow in order to take advantages of its powerful capability of computation acceleration, or, just install CPU version if not so. To be more specific, the CUDA Computing Capability of your graphics card that you can check on NVIDIA official website should not be less than 3.0.
2. Install the Python environment. Anaconda is recommended. It is an open-source release version of Python that provides a full environment for scientific computation including common libraries such as NumPy and SciPy, or you can choose your favorite ones of course. Note that TensorFlow only supports Python Ver 3.X under Windows when we write this handbook.
   - You can choose to add the directory of Anaconda into the PATH (though not recommended by the installation wizard). It enables you to call all Anaconda commands under command line or Powershell directly. You can always call them under the Anaconda Prompt started in the Start Menu.
3. (For GPU version installation) Install the NVIDIA graphics driver, CUDA Toolkit and cuDNN. You should note that:
   - We recommend you install it through the following order: 1) latest NVIDIA graphics driver 2) CUDA (without selecting the built-in driver when installing since the built-in ones may be out-of-date) 3) cuDNN;
   - There is a quite simple way to install drivers in Ubuntu. First click "Software & Updates" in "System Setting", then toggle on "Using NVIDIA binary driver" option in "Additional Drivers" and click "Apply Changes" for system to install NVIDIA drivers automatically, otherwise, it won't be peaceful for NVIDIA installation on Linux. You should disable the built-in graphics driver Nouveau and Secure Boot function of the motherboard. You can seek a more detailed guidance here;
   - The version of CUDA Toolkit and cuDNN must agree with the requirements on TensorFlow official website which does not always require the latest version.
   - You have to copy the downloaded files of cuDNN to the installation directory of CUDA to complete cuDNN installation.

2.2.2 Install

These are the following steps of TensorFlow installation under Anaconda (taking Windows as example):

1. Create a conda environment named `tensorflow`

```
conda create -n tensorflow python=X.X # Substitute "X.X" with your own Python version, e.g. "3.6".
```

2. Activate the environment

```
activate tensorflow
```

3. Use pip to install TensorFlow

Install the CPU version

```
pip install tensorflow
```

Or, install the GPU version

```
pip install tensorflow-gpu
```

You can also choose to install the Nightly version of TensorFlow if you want. This version may include some latest features compared to the official version (e.g. the Eager Execution mode in this handbook was only supported in the Nightly version before TensorFlow Ver 1.8) yet with some instability. You can do so by running `pip install tf-nightly` (CPU version) or `pip install tf-nightly-gpu` (GPU version) in a new virtual environment. If you are going to install the GPU version, it may require higher versions of CUDA and cuDNN. Fortunately different versions of CUDA and cuDNN can coexist.

If it is slow to install through pip command in China, you may want to try TensorFlow mirror on TUNA.

2.3 First Program

We write a piece of code to verify the installation.

Enter `activate tensorflow` under command line to enter the previously built conda environment with TensorFlow. Then enter `python` to enter Python environment. Input the following codes line by line:

```
import tensorflow as tf
tf.enable_eager_execution()

A = tf.constant([[1, 2], [3, 4]])
B = tf.constant([[5, 6], [7, 8]])
C = tf.matmul(A, B)

print(C)
```

If the output is:

```
tf.Tensor(
[[19 22]
[43 50]], shape=(2, 2), dtype=int32)
```

We can draw conclusions that TensorFlow was successfully installed. It' s normal for the program to output some prompt messages when running.

Here we use Python. You can get Python tutorials on https://docs.python.org/3/tutorial/. From now on we assume that readers are familiar with the basics of Python. Relax, Python is easy to handle and advanced features of Python will be barely involved in TensorFlow. We recommend you to use PyCharm as your Python IDE. If you are a student with an email address ended with .edu, you can apply for a free license here <http://www.jetbrains.com/student/>'_. You can always download PyCharm Community version whose main functions do not differ that much from the former if you do not meet the aforementioned criteria.

CHAPTER 3

TensorFlow Basic

This chapter describes basic operations in TensorFlow.

Prerequisites:

- Basic Python operations (assignment, branch & loop statement, library import)
- ‘With’ statement in Python
- NumPy , a common library for scientific computation, important for TensorFlow
- Vectors & Matrices operations (matrix addition & subtraction, matrix multiplication with vectors & matrices, matrix transpose, etc., Quiz: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$ =?)
- Derivatives of functions , derivatives of multivariable functions (Quiz: $f(x, y) = x^2 + xy + y^2$, $\frac{\partial f}{\partial x} =$?, $\frac{\partial f}{\partial y}$ =?)
- Linear regression;
- Gradient descent that searches local minima of a function

3.1 TensorFlow 1+1

TensorFlow can be simply regarded as a library of scientific calculation (like Numpy in Python). Here we calculate $1 + 1$ and $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$ as our first example.

```
import tensorflow as tf
tf.enable_eager_execution()

a = tf.constant(1)
b = tf.constant(1)
c = tf.add(a, b)    # The expression c = a + b is equivalent.

print(c)

A = tf.constant([[1, 2], [3, 4]])
B = tf.constant([[5, 6], [7, 8]])
C = tf.matmul(A, B)

print(C)
```

Output:

```
tf.Tensor(2, shape=(), dtype=int32)
tf.Tensor(
[[19 22]
[43 50]], shape=(2, 2), dtype=int32)
```

The code above declares four **tensors** named `a`, `b`, `A` and `B`. It also invokes two **operations** `tf.add()` and `tf.matmul()` which respectively do addition and matrix multiplication on tensors. Operation results are immediately stored in the tensors `c` and `C`. **Shape** and **dtype** are two major attributes of a tensor. Here `a`, `b` and `c` are scalars with null shape and int32 dtype, while `A`, `B`, `C` are 2-by-2 matrices with `(2, 2)` shape and int32 dtype.

In machine learning, it' s common to differentiate functions. TensorFlow provides us with the powerful **Automatic Differentiation Mechanism** for differentiation. The following codes show how to utilize `tf.GradientTape()` to get the slope of $y(x) = x^2$ at $x = 3$.

```
import tensorflow as tf
tf.enable_eager_execution()

x = tf.get_variable('x', shape=[1], initializer=tf.constant_initializer(3.))
with tf.GradientTape() as tape:     # All steps in the context of tf.GradientTape() are recorded␣
↪for differentiation.
    y = tf.square(x)
y_grad = tape.gradient(y, x)        # Differentiate y with respect to x.
print([y.numpy(), y_grad.numpy()])
```

Output:

```
[array([9.], dtype=float32), array([6.], dtype=float32)]
```

Here `x` is a **variable** initialized to 3, declared by `tf.get_variable()`. Like common tensors, variables also have shape and dtype attributes, but require an initialization. We can assign an initializer to `tf.get_variable()` by setting the `Initializer` parameter. Here we use `tf.constant_initializer(3.)` to initialize the variable `x` to `3.` with a float32 dtype.[1]. An important difference between variables and common tensors is that a function can be differentiated by variables, not by tensors, using the automatic differentiation mechanism by default. Therefore variables are usually used as parameters defined in machine learning models. `tf.GraidentTape()` is a recorder of automatic differentiation which records all variables and steps of calculation automatically. In the previous example, the variable `x` and the calculation step `y = tf.square(x)` are recorded automatically, thus the derivative of the tensor `y` with respect to `x` can be obtained through `y_grad = tape.gradient(y, x)`.

In machine learning, calculating the derivatives of a multivariable function, a vector or a matrix is a more common case, which is a piece cake for TensorFlow. The following codes show how to utilize `tf.GradientTape()` to differentiate $L(w, b) = \|Xw + b - y\|^2$ with respect to w and b at $w = (1, 2)^T, b = 1$.

```
X = tf.constant([[1., 2.], [3., 4.]])
y = tf.constant([[1.], [2.]])
w = tf.get_variable('w', shape=[2, 1], initializer=tf.constant_initializer([[1.], [2.]]))
b = tf.get_variable('b', shape=[1], initializer=tf.constant_initializer([1.]))
with tf.GradientTape() as tape:
    L = 0.5 * tf.reduce_sum(tf.square(tf.matmul(X, w) + b - y))
w_grad, b_grad = tape.gradient(L, [w, b])        # Differentiate L with respect to w and b.
print([L.numpy(), w_grad.numpy(), b_grad.numpy()])
```

Output:

```
[62.5, array([[35.],
   [50.]], dtype=float32), array([15.], dtype=float32)]
```

Here the operation `tf.square()` squares every element in the input tensor without altering its shape. The operation `tf.reduce_sum()` outputs the sum of all elements in the input tensor with a null shape (the dimensions of the summation can be indicated by the `axis` parameter, while all elements are summed up if not specified). TensorFlow contains a large number of tensor operation APIs including mathematical operations, tensor shape operations (like `tf.reshape()`), slicing and concatenation (like `tf.concat()`), etc. You can heck TensorFlow official API documentation[2] for further information.

[1] We can add a decimal point after an integer to make it become a floating point number in Python. E.g. `3.` represents the floating point number `3.0`.

[2] Mainly refer to Tensor Transformations and Math. Note that the tensor operation API of TensorFlow is very similar to Numpy, thus one can get started on TensorFlow rather quickly if knowing about the latter.

As we can see from the output, TensorFlow helps us figure out that

$$L((1,2)^T, 1) = 62.5$$
$$\frac{\partial L(w,b)}{\partial w}|_{w=(1,2)^T,b=1} = \begin{bmatrix} 35 \\ 50 \end{bmatrix}$$
$$\frac{\partial L(w,b)}{\partial b}|_{w=(1,2)^T,b=1} = 15$$

3.2 A Basic Example: Linear Regression

Let' s consider a practical problem. The house prices of a city between 2013 and 2017 are given by the following table:

Year	2013	2014	2015	2016	2017
Price	12000	14000	15000	16500	17500

Now we want to do linear regression on the given data, i.e. using the linear model $y = ax + b$ to fit the data, where `a` and `b` are unknown parameters.

First, we define and normalize the data.

```python
import numpy as np

X_raw = np.array([2013, 2014, 2015, 2016, 2017])
y_raw = np.array([12000, 14000, 15000, 16500, 17500])

X = (X_raw - X_raw.min()) / (X_raw.max() - X_raw.min())
y = (y_raw - y_raw.min()) / (y_raw.max() - y_raw.min())
```

Then, we use gradient descent to evaluate these two parameters `a` and `b` in the linear model[3].

Recalling from the fundamentals of machine learning, for searching local minima of a multivariable function $f(x)$, we use gradient descent which taking the following steps:

- Initialize the argument to x_0 and have $k = 0$
- Iterate the following steps repeatedly till the convergence criteria is met:
  - Find the gradient of the function $f(x)$ with respect to the parameter $\nabla f(x_k)$
  - Update the parameter $x_{k+1} = x_k - \gamma \nabla f(x_k)$ where γ is the learning rate (like the step size of the gradient descent)
  - $k \leftarrow k + 1$

[3] In fact there is an analytic solution for the linear regression. We use gradient descent here just for showing you how TensorFlow works.

Next we focus on how to implement gradient descent in order to solve the linear regression $\min_{a,b} L(a,b) = \sum_{i=1}^{n}(ax_i + b - y_i)^2$.

3.2.1 NumPy

The implementation of machine learning models is not a patent of TensorFlow. In fact, even most common scientific calculators or tools can solve simple models. Here, we use Numpy, a general library for scientific computation, to implement gradient descent. Numpy supports multidimensional arrays to represent vectors, matrices and tensors with more dimensions. Meanwhile, it also supports lots of operations on multidimensional arrays (e.g. `np.dot()` calculates the inner products and `np.sum()` adds up all the elements). In this way Numpy is somewhat like MATLAB. In the following codes, we evaluate the partial derivatives of loss function with respect to the parameters `a` and `b` manually[4], and then iterate by gradient descent to acquire the value of `a` and `b` eventually.

```
a, b = 0, 0

num_epoch = 10000
learning_rate = 1e-3
for e in range(num_epoch):
    # Calculate the gradient of the loss function with respect to arguments (model parameters)␣
↪manually.
    y_pred = a * X + b
    grad_a, grad_b = (y_pred - y).dot(X), (y_pred - y).sum()

    # Update parameters.
    a, b = a - learning_rate * grad_a, b - learning_rate * grad_b

print(a, b)
```

However, you may have noticed that there are several pain points using common libraries for scientific computation to implement machine learning models:

- It’s often inevitable to differentiate functions manually. Simple ones may be fine, however the more complex ones (especially commonly appeared in deep learning models) are another story. Manual differentiation may be painful, even infeasible in the latter cases.
- It’s also often inevitable to update parameters based on the gradients manually. Manual update is still easy here because the gradient descent is a rather basic method while it’s not going to be easy anymore if we apply a more complex approach to update parameters (like Adam or Adagrad).

However, the appearance of TensorFlow eliminates these pain points to a large extent, granting users convenience for implementing machine learning models.

[4] The loss function here is the mean square error $L(x) = \frac{1}{2}\sum_{i=1}^{5}(ax_i + b - y_i)^2$ whose partial derivatives with respect to `a` and `b` are $\frac{\partial L}{\partial a} = \sum_{i=1}^{5}(ax_i + b - y)x_i$ and $\frac{\partial L}{\partial b} = \sum_{i=1}^{5}(ax_i + b - y)$.

3.2.2 TensorFlow

The **Eager Execution Mode** of TensorFlow[5] have very similar operations as the above-mentioned Numpy. In addition, it also provides us with a series of critical functions for deep learning such as faster operation speed (need support from GPU), automatic differentiation and optimizers, etc. We will show how to do linear regression using Tensorflow. You may notice that its code structure is similar to the one of Numpy. Here we delegates TensorFlow to do two important jobs:

- Using `tape.gradient(ys, xs)` to get the gradients automatically;
- Using `optimizer.apply_gradients(grads_and_vars)` to update parameters automatically.

```
X = tf.constant(X)
y = tf.constant(y)

a = tf.get_variable('a', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)
b = tf.get_variable('b', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)
variables = [a, b]

num_epoch = 10000
optimizer = tf.train.GradientDescentOptimizer(learning_rate=1e-3)
for e in range(num_epoch):
    # Use tf.GradientTape() to record the gradient info of the loss function
    with tf.GradientTape() as tape:
        y_pred = a * X + b
        loss = 0.5 * tf.reduce_sum(tf.square(y_pred - y))
    # TensorFlow calculates the gradients of the loss function with respect to each argument␣
↪(model paramter) automatically.
    grads = tape.gradient(loss, variables)
    # TensorFlow updates parameters automatically based on gradients.
    optimizer.apply_gradients(grads_and_vars=zip(grads, variables))
```

Here, we use the aforementioned approach to calculate the partial derivatives of the loss function with respect to each parameter, while we also use `tf.train.GradientDescentOptimizer(learning_rate=1e-3)` to declare an **optimizer** for graident descent with a learning rate of 1e-3. The optimizer can help us update parameters based on the result of differentiation in order to minimize a specific loss function by calling its `apply_gradients()` interface.

Note that, for calling `optimizer.apply_gradients()` to update model parameters, we need to provide it with parameters `grads_and_vars`, i.e. the variables to be updated (like `variables` in the aforementioned codes). To be specific, a Python list has to be passed, whose every element is a (partial derivative with respect to a variable, this variable) pair. For instance, `[(grad_w, w), (grad_b, b)]` is passed here. By executing `grads = tape.gradient(loss, variables)` we get partial derivatives of the loss function with

[5] The opposite of Eager Execution is Graph Execution that TensorFlow adopts before version 1.8 in Mar 2018. This handbook is mainly written for Eager Execution aiming at fast iterative development, however the basic usage of Graph Execution is also attached in the appendices in case of reference.

respect to each variable recorded in `tape`, i.e. `grads = [grad_w, grad_b]`. Then we use `zip()` in Python to pair the elements in `grads = [grad_w, grad_b]` and `vars = [w, b]` together respectively so as to get the required parameters.

In practice, we usually build much more complex models rather the linear model `y_pred = tf.matmul(X, w) + b` here which can be simply written in a single line. Therefore, we often write a model class and call it by `y_pred = model(X)` when needed. *The following chapter* elaborates writing model classes.

CHAPTER 4

TensorFlow Models

This chapter describes how to build a dynamic model with TensorFlow quickly.

Prerequisites:

- Python OOP (definition of classes & methods in Python, class inheritance, construction and destruction functions, using super() to call methods of the parent class, using __call__() to call an instance, etc.)
- Multilayer perceptrons, convolutional neural networks, recurrent neural networks and reinforcement learning (references given before each chapter).

4.1 Models & Layers

As mentioned in the last chapter, to improve the reusability of codes, we usually implement models as classes and use statements like `y_pred() = model(X)` to call the models. The structure of a **model class** is rather simple which basically includes `__init__()` (for construction and initialization) and `call(input)` (for model invocation) while you can also define your own methods if necessary.[1]

```
class MyModel(tf.keras.Model):
    def __init__(self):
        super().__init__()     # Use super(MyModel, self).__init__() under Python 2.
```

[1] In Python classes, calling instances of the class `myClass` by such as `myClass()` is equivalent to `myClass.__call__()`. Here our model inherits from the parent class `tf.keras.Model`. This class includes the definition of `__call__()`, calling `call()` and also doing some internal keras operations. By inheriting from `tf.keras.Model` and overloading its `call()` method, we can add the codes for calling while keep the structure of keras, for details please refer to the `__call__()` part of the prerequisites in this chapter.

```
        # Add initialization code here (including layers to be used in the "call" method).

    def call(self, inputs):
        # Add model invocation code here (processing inputs and returning outputs).
        return output
```

Here our model inherits from `tf.keras.Model`. Keras is an advanced neural network API written in Python and is now supported by and built in official TensorFlow. One benefit of inheriting from `tf.keras.Model` is that we will be able to use its several methods and attributes like acquiring all variables in the model through the `model.variables` attribute after the class is instantiated, which saves us from indicating variables one by one explicitly.

Meanwhile, we introduce the concept of **layers**. Layers can be regarded as finer components encapsulating the computation procedures and variables compared with models. We can use layers to build up models quickly.

The simple linear model `y_pred = tf.matmul(X, w) + b` mentioned in the last chapter can be implemented through model classes:

```
import tensorflow as tf
tf.enable_eager_execution()

X = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
y = tf.constant([[10.0], [20.0]])


class Linear(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.dense = tf.keras.layers.Dense(units=1, kernel_initializer=tf.zeros_initializer(),
            bias_initializer=tf.zeros_initializer())

    def call(self, input):
        output = self.dense(input)
        return output


# The structure of the following codes is similar to the previous one.
model = Linear()
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
for i in range(100):
    with tf.GradientTape() as tape:
        y_pred = model(X)      # Call the model.
        loss = tf.reduce_mean(tf.square(y_pred - y))
    grads = tape.gradient(loss, model.variables)
    optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))
print(model.variables)
```

Here we didn' t explicitly declare two variables `W` and `b` or write the linear transformation `y_pred = tf.matmul(X, w) + b`, but instead instantiated a densely-connected layer (`tf.keras.layers.Dense`) in the initialization and called this layer in the "call" method. The densely-connected layer encapsulates the `output = activation(tf.matmul(input, kernel) + bias)` linear transformation, the activation function, and two variables `kernel` and `bias`. This densely-connected layer would be equivalent to the aforementioned linear transformation if the activation function is not specified (i.e. `activation(x) = x`). By the way, the densely-connected layer may be the most frequent in our daily model coding.

Please refer to Custom Layers if you need to declare variables explicitly and customize operations.

4.2 A Basic Example: Multilayer Perceptrons (MLP)

Let' s begin with implementing a simple Multilayer Perceptron as an introduction of writing models in TensorFlow. Here we use the multilayer perceptron to classify the MNIST handwriting digit image dataset.

Fig. 4.1: An image example of MNIST handwriting digits.

Before the main course, we implement a simple `DataLoader` class for loading the MNIST dataset.

```
class DataLoader():
    def __init__(self):
        mnist = tf.contrib.learn.datasets.load_dataset("mnist")
        self.train_data = mnist.train.images                                 # np.array [55000,␣
↪784].
        self.train_labels = np.asarray(mnist.train.labels, dtype=np.int32)   # np.array [55000] of␣
↪int32.
        self.eval_data = mnist.test.images                                   # np.array [10000,␣
↪784].
        self.eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)     # np.array [10000] of␣
↪int32.

    def get_batch(self, batch_size):
        index = np.random.randint(0, np.shape(self.train_data)[0], batch_size)
        return self.train_data[index, :], self.train_labels[index]
```

The implementation of the multilayer perceptron model class is similar to the aforementioned linear model class except the former has more layers (as its name "multilayer" suggests) and introduces a nonlinear activation function (here ReLU function is used by the code `activation=tf.nn.relu` below). This model receives a vector (like a flattened 1*784 handwriting digit image here) and outputs a 10 dimensional signal representing the probability of this image being 0 to 9, respectively. Here we introduce a "predict" method which predicts the handwriting digits. It chooses the digit with the maximum likelihood as an output.

```
class MLP(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.dense1 = tf.keras.layers.Dense(units=100, activation=tf.nn.relu)
        self.dense2 = tf.keras.layers.Dense(units=10)

    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
        return x

    def predict(self, inputs):
        logits = self(inputs)
        return tf.argmax(logits, axis=-1)
```

Define some hyperparameters for the model:

```
num_batches = 1000
batch_size = 50
learning_rate = 0.001
```

Instantiate the model, the data loader class and the optimizer:

```
model = MLP()
data_loader = DataLoader()
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

And iterate the following steps:

- Randomly read a set of training data from DataLoader;
- Feed the data into the model to acquire its predictions;
- Compare the predictions with the correct answers to evaulate the loss function;
- Differentiate the loss function with respect to model parameters;
- Update the model parameters in order to minimize the loss.

The code implementation is as follows:

```python
for batch_index in range(num_batches):
    X, y = data_loader.get_batch(batch_size)
    with tf.GradientTape() as tape:
        y_logit_pred = model(tf.convert_to_tensor(X))
        loss = tf.losses.sparse_softmax_cross_entropy(labels=y, logits=y_logit_pred)
        print("batch %d: loss %f" % (batch_index, loss.numpy()))
    grads = tape.gradient(loss, model.variables)
    optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))
```

Then, we will use the validation dataset to examine the performance of this model. To be specific, we will compare the predictions with the answers on the validation dateset, and output the ratio of correct predictions:

```python
num_eval_samples = np.shape(data_loader.eval_labels)[0]
y_pred = model.predict(data_loader.eval_data).numpy()
print("test accuracy: %f" % (sum(y_pred == data_loader.eval_labels) / num_eval_samples))
```

Output:

```
test accuracy: 0.947900
```

Note that we can easily get an accuracy of 95% with even a simple model like this.

4.3 Convolutional Neural Networks (CNN)

The Convolutional Neural Network is an artificial neural network resembled as the animal visual system . It consists of one or more convolutional layers, pooling layers and dense layers. For detailed theories please refer to the Convolutional Neural Network chapter of the *Machine Learning* course by Professor Li Hongyi of National Taiwan University.

The specific implementation is as follows, which is very similar to MLP except some new convolutional layers and pooling layers.

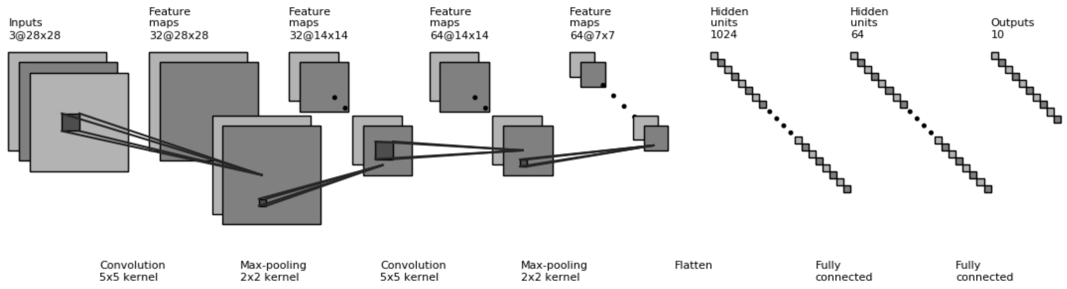


Fig. 4.2: Figure of the CNN structure

```
class CNN(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.conv1 = tf.keras.layers.Conv2D(
            filters=32,             # Numbers of convolution kernels.
            kernel_size=[5, 5],     # Size of the receptive field.
            padding="same",         # Padding strategy.
            activation=tf.nn.relu   # Activation function.
        )
        self.pool1 = tf.keras.layers.MaxPool2D(pool_size=[2, 2], strides=2)
        self.conv2 = tf.keras.layers.Conv2D(
            filters=64,
            kernel_size=[5, 5],
            padding="same",
            activation=tf.nn.relu
        )
        self.pool2 = tf.keras.layers.MaxPool2D(pool_size=[2, 2], strides=2)
        self.flatten = tf.keras.layers.Reshape(target_shape=(7 * 7 * 64,))
        self.dense1 = tf.keras.layers.Dense(units=1024, activation=tf.nn.relu)
        self.dense2 = tf.keras.layers.Dense(units=10)

    def call(self, inputs):
        inputs = tf.reshape(inputs, [-1, 28, 28, 1])
        x = self.conv1(inputs)                  # [batch_size, 28, 28, 32].
        x = self.pool1(x)                       # [batch_size, 14, 14, 32].
        x = self.conv2(x)                       # [batch_size, 14, 14, 64].
        x = self.pool2(x)                       # [batch_size, 7, 7, 64].
        x = self.flatten(x)                     # [batch_size, 7 * 7 * 64].
        x = self.dense1(x)                      # [batch_size, 1024].
        x = self.dense2(x)                      # [batch_size, 10].
        return x

    def predict(self, inputs):
        logits = self(inputs)
```

By substituting `model = MLP()` in the last chapter with `model = CNN()`, we get the following output:

```
test accuracy: 0.988100
```

We can see that there is a significant improvement of accuracy. In fact, the accuracy can be improved further by altering the network structure of the model (e.g. adding the Dropout layer to avoid overfitting).

4.4 Recurrent Neural Networks (RNN)

The Recurrent Neural Network is a kind of neural network suitable for processing sequential data, which is generally used for language modeling, text generation and machine translation, etc. For RNN theories, please refer to:

- Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs.
- Recurrent Neural Network (part 1) and Recurrent Neural Network (part 2) from the *Machine Learning* course by Professor Li Hongyi of Taiwan University.
- The principles of LSTM: Understanding LSTM Networks.
- Generation of RNN sequences: [Graves2013].

Here we use RNN to generate a piece of Nietzsche style text.[2]

This task essentially predicts the probability distribution of the following alphabet of a given English text segment. For example, we have the following sentence:

```
I am a studen
```

This sentence (sequence) has 13 characters (including spaces) in total. Based on our experience, we can predict the following alphabet will probably be "t" when we read this sequence. We would like to build a model that receives num_batch sequences consisting of seq_length encoded characters and the input tensor with shape [num_batch, seq_length], and outputs the probability distribution of the following character with the dimension of num_chars (number of characters) and the shape [num_batch, num_chars]. We will sample from the probability distribution of the following character as a prediction and generate the second, third following characters step by step in order to complete the task of text generation.

First, we still implement a simple `DataLoader` class to read text and encode it in characters.

```
class DataLoader():
    def __init__(self):
        path = tf.keras.utils.get_file('nietzsche.txt',
            origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
        with open(path, encoding='utf-8') as f:
            self.raw_text = f.read().lower()
        self.chars = sorted(list(set(self.raw_text)))
        self.char_indices = dict((c, i) for i, c in enumerate(self.chars))
        self.indices_char = dict((i, c) for i, c in enumerate(self.chars))
        self.text = [self.char_indices[c] for c in self.raw_text]

    def get_batch(self, seq_length, batch_size):
        seq = []
```

[2] The task and its implementation are referred to https://github.com/keras-team/keras/blob/master/examples/lstm_text_generation.py

```
        next_char = []
        for i in range(batch_size):
            index = np.random.randint(0, len(self.text) - seq_length)
            seq.append(self.text[index:index+seq_length])
            next_char.append(self.text[index+seq_length])
        return np.array(seq), np.array(next_char)       # [num_batch, seq_length], [num_batch].
```

Then we implement the model. We instantiate a common `BasicLSTMCell` unit and a dense layer for linear transformation in the `__init__` method. We first do an One-Hot operation on the sequence, i.e. transforming the code i into an n dimensional vector with the value 1 on the i-th position and value 0 elsewhere. Here n is the number of characters. The shape of the transformed sequence tensor is [num_batch, seq_length, num_chars]. After that, we will feed the sequences one by one into the RNN unit, i.e. feeding the state of the RNN unit `state` and sequences `inputs[:, t, :]` of the current time t into the RNN unit, and get the output of the current time `output` and the RNN unit state of the next time t+1. We take the last output of the RNN unit, transform it into num_chars dimensional one through a dense layer, and consider it as the model output.

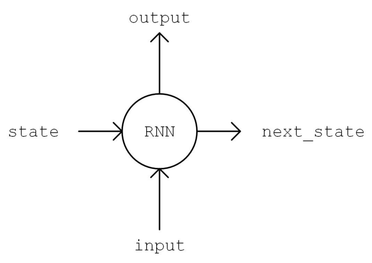


Fig. 4.3: Figure of `output, state = self.cell(inputs[:, t, :], state)`

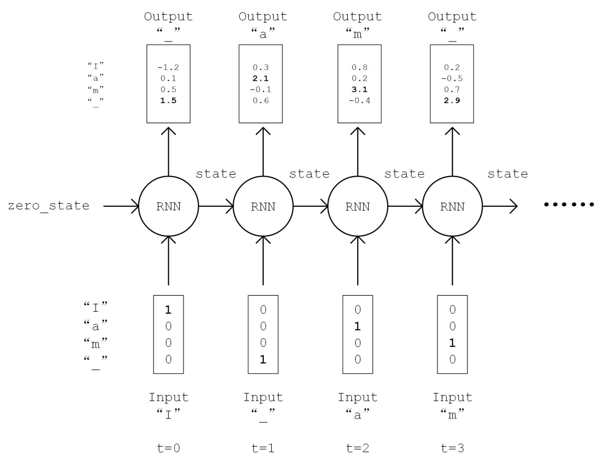


Fig. 4.4: Figure of RNN process

The implementation is as follows:

```
class RNN(tf.keras.Model):
    def __init__(self, num_chars):
        super().__init__()
        self.num_chars = num_chars
        self.cell = tf.nn.rnn_cell.BasicLSTMCell(num_units=256)
        self.dense = tf.keras.layers.Dense(units=self.num_chars)

    def call(self, inputs):
        batch_size, seq_length = tf.shape(inputs)
        inputs = tf.one_hot(inputs, depth=self.num_chars)       # [batch_size, seq_length, num_
↪chars].
        state = self.cell.zero_state(batch_size=batch_size, dtype=tf.float32)
        for t in range(seq_length.numpy()):
            output, state = self.cell(inputs[:, t, :], state)
        output = self.dense(output)
        return output
```

The training process is almost the same with the previous chapter, which we reiterate here:

- Randomly read a set of training data from DataLoader;
- Feed the data into the model to acquire its predictions;
- Compare the predictions with the correct answers to evaluate the loss function;
- Differentiate the loss function with respect to model parameters;
- Update the model parameters in order to minimize the loss.

```
data_loader = DataLoader()
model = RNN(len(data_loader.chars))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
for batch_index in range(num_batches):
    X, y = data_loader.get_batch(seq_length, batch_size)
    with tf.GradientTape() as tape:
        y_logit_pred = model(X)
        loss = tf.losses.sparse_softmax_cross_entropy(labels=y, logits=y_logit_pred)
        print("batch %d: loss %f" % (batch_index, loss.numpy()))
    grads = tape.gradient(loss, model.variables)
    optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))
```

There is one thing we should notice about the process of text generation is that earlier we used the `tf.argmax()` function to regard the value with the maximum likelihood as the prediction. However, this method of prediction will be too rigid for text generation which also deprives the richness of the generated text. Thus, we use the `np.random.choice()` function for sampling based on the generated probability distribution by which even characters with small likelihood can still be possible to be sampled. Meanwhile we introduce the `temperature` parameter to control the shape of the distribution. Larger the value, flatter the distribution (smaller difference between the maximum and the minimum) and richer the generated text. Vice versa.

```
    def predict(self, inputs, temperature=1.):
        batch_size, _ = tf.shape(inputs)
        logits = self(inputs)
        prob = tf.nn.softmax(logits / temperature).numpy()
        return np.array([np.random.choice(self.num_chars, p=prob[i, :])
                         for i in range(batch_size.numpy())])
```

We can get the generated text by this step-by-step continuing prediction.

```
X_, _ = data_loader.get_batch(seq_length, 1)
for diversity in [0.2, 0.5, 1.0, 1.2]:
    X = X_
    print("diversity %f:" % diversity)
    for t in range(400):
        y_pred = model.predict(X, diversity)
        print(data_loader.indices_char[y_pred[0]], end='', flush=True)
        X = np.concatenate([X[:, 1:], np.expand_dims(y_pred, axis=1)], axis=-1)
```

The generated text is as follows:

```
diversity 0.200000:
conserted and conseive to the conterned to it is a self--and seast and the selfes as a seast the␣
↪expecience and and and the self--and the sered is a the enderself and the sersed and as a the␣
↪concertion of the series of the self in the self--and the serse and and the seried enes and␣
↪seast and the sense and the eadure to the self and the present and as a to the self--and the␣
↪seligious and the enders

diversity 0.500000:
can is reast to as a seligut and the complesed
has fool which the self as it is a the beasing and us immery and seese for entoured underself of␣
↪the seless and the sired a mears and everyther to out every sone thes and reapres and seralise␣
↪as a streed liees of the serse to pease the cersess of the selung the elie one of the were as we␣
↪and man one were perser has persines and conceity of all self-el

diversity 1.000000:
entoles by
their lisevers de weltaale, arh pesylmered, and so jejurted count have foursies as is
descinty iamo; to semplization refold, we dancey or theicks-welf--atolitious on his
such which
here
oth idey of pire master, ie gerw their endwit in ids, is an trees constenved mase commars is leed␣
↪mad decemshime to the mor the elige. the fedies (byun their ope wopperfitious--antile and the it␣
↪as the f
```

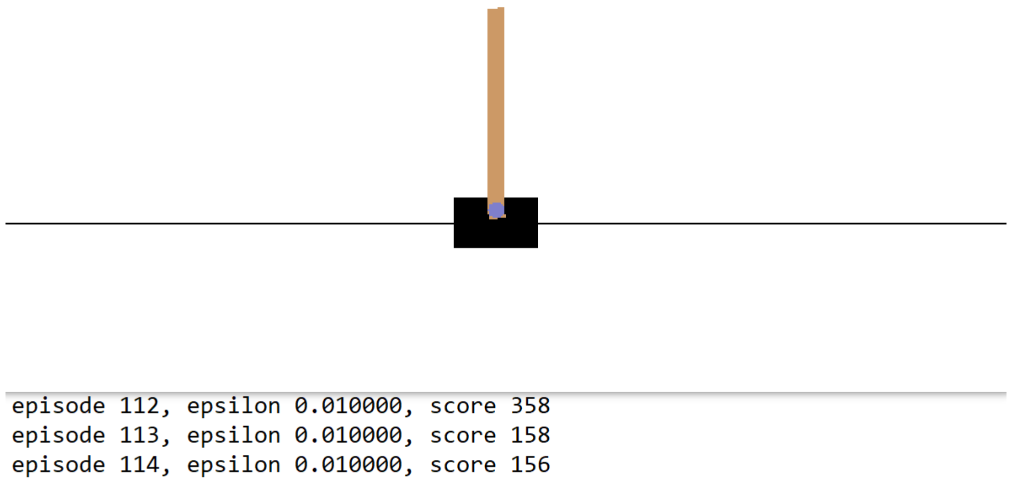
```
diversity 1.200000:
cain, elvotidue, madehoublesily
inselfy!--ie the rads incults of to prusely le]enfes patuateded:.--a coud--theiritibaior
↪"nrallysengleswout peessparify oonsgoscess teemind thenry ansken suprerial mus, cigitioum: 4reas.
↪ whouph: who
eved
arn inneves to sya" natorne. hag open reals whicame oderedte,[fingo is
zisternethta simalfule dereeg hesls lang-lyes thas quiin turjentimy; periaspedey tomm--whach
```

4.5 Deep Reinforcement Learning (DRL)

The Reinforcement Learning focuses on how to act in the environment in order to maximize the expected benefits. It becomes more powerful if combined with the deep learning technique. The recently famous AlphaGo is a typical application of the deep reinforcement learning. For its basic knowledge please refer to:

- Demystifying Deep Reinforcement Learning.
- [Mnih2013].

Here, we use the deep reinforcement learning to learn how to play CartPole. To be simple, our model needs to control the pole to keep it in a straight balance state.



```
episode 112, epsilon 0.010000, score 358
episode 113, epsilon 0.010000, score 158
episode 114, epsilon 0.010000, score 156
```

Fig. 4.5: The CartPole game

We use the CartPole game environment in the Gym Environment Library powered by OpenAI. Please refer to the official documentation and here for detailed installation steps and tutorials.

```
import gym

env = gym.make('CartPole-v1')       # Instantiate a game environment. The parameter is its name.
state = env.reset()                 # Initialize the environment and get its initial state.
while True:
    env.render()                    # Render the current frame.
    action = model.predict(state)   # Assume we have a trained model that can predict the action␣
↪based on the current state.
    next_state, reward, done, info = env.step(action)   # Let the environment to execute the␣
↪action, get the next state, the reward of the action, a flag whether the game is done, and extra␣
↪information.
    if done:                        # Exit if the game is done.
        break
```

Therefore our task is to train a model that can predict good actions based on the current state. Roughly speaking, a good action should maximize the sum of rewards gained during the whole game process, which is also the target of reinforcement learning.

The following code shows how to use the Deep Q-Learning in deep reinforcement learning to train the model.

```
import tensorflow as tf
import numpy as np
import gym
import random
from collections import deque

tf.enable_eager_execution()
num_episodes = 500
num_exploration_episodes = 100
max_len_episode = 1000
batch_size = 32
learning_rate = 1e-3
gamma = 1.
initial_epsilon = 1.
final_epsilon = 0.01


# Q-network is used to fit Q function resemebled as the aforementioned multilayer perceptron. It␣
↪inputs state and output Q-value under each action (2 dimensional under CartPole).
class QNetwork(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.dense1 = tf.keras.layers.Dense(units=24, activation=tf.nn.relu)
        self.dense2 = tf.keras.layers.Dense(units=24, activation=tf.nn.relu)
        self.dense3 = tf.keras.layers.Dense(units=2)
    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
        x = self.dense3(x)
        return x

    def predict(self, inputs):
        q_values = self(inputs)
        return tf.argmax(q_values, axis=-1)


env = gym.make('CartPole-v1')       # Instantiate a game environment. The parameter is its name.
model = QNetwork()
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
replay_buffer = deque(maxlen=10000)
epsilon = initial_epsilon
for episode_id in range(num_episodes):
    state = env.reset()             # Initialize the environment and get its initial state.
    epsilon = max(
        initial_epsilon * (num_exploration_episodes - episode_id) / num_exploration_episodes,
        final_epsilon)
    for t in range(max_len_episode):
        env.render()                # Render the current frame.
        if random.random() < epsilon:               # Epsilon-greedy exploration strategy.
            action = env.action_space.sample()      # Choose random action with the probability of␣
↪epilson.
        else:
            action = model.predict(
                tf.constant(np.expand_dims(state, axis=0), dtype=tf.float32)).numpy()
            action = action[0]
        next_state, reward, done, info = env.step(action)               # Let the environment to␣
↪execute the action, get the next state of the action, the reward of the action, whether the game␣
↪is done and extra information.
        reward = -10. if done else reward                               # Give a large negative␣
↪reward if the game is over.
        replay_buffer.append((state, action, reward, next_state, done)) # Put the (state, action,␣
↪reward, next_state) quad back into the experience replay pool.
        state = next_state

        if done:                                                        # Exit this round and␣
↪enter the next episode if the game is over.
            print("episode %d, epsilon %f, score %d" % (episode_id, epsilon, t))
            break
        if len(replay_buffer) >= batch_size:
            batch_state, batch_action, batch_reward, batch_next_state, batch_done = \
                [np.array(a, dtype=np.float32) for a in zip(*random.sample(replay_buffer, batch_
↪size))] # Randomly take a batch quad from the experience replay pool.
            q_value = model(tf.constant(batch_next_state, dtype=tf.float32))
            y = batch_reward + (gamma * tf.reduce_max(q_value, axis=1)) * (1 - batch_done)  #␣
↪Calculate y according to the method in the paper.
            with tf.GradientTape() as tape:
                loss = tf.losses.mean_squared_error(        # Minimize the distance between y and␣
↪Q-value.
                    labels=y,
                    predictions=tf.reduce_sum(model(tf.constant(batch_state)) *
                                              tf.one_hot(batch_action, depth=2), axis=1)
                )
            grads = tape.gradient(loss, model.variables)
            optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))       #␣
↪Calculate the gradient and update parameters.
```

4.6 Custom Layers *

Maybe you want to ask that what if these layers can’ t satisfy my needs and I need to customize my own layers?

In fact, we can not only inherit from `tf.keras.Model` to write your own model class, but also inherit from `tf.keras.layers.Layer` to write your own layer.

```
class MyLayer(tf.keras.layers.Layer):
    def __init__(self):
        super().__init__()
        # Initialization.

    def build(self, input_shape):     # input_shape is a TensorFlow class object which provides␣
↪the input shape.
        # Call this part when the layer is firstly used. Variables created here will have self-
↪adapted shapes without specification from users. They can also be created in __init__ part if␣
↪their shapes are already determined.
        self.variable_0 = self.add_variable(...)
        self.variable_1 = self.add_variable(...)

    def call(self, input):
        # Model calling (process inputs and return outputs).
        return output
```

For example, if we want to implement a dense layer in the first section of this chapter with a specified output dimension of 1, we can write as below, creating two variables in the `build` method and do operations on them:

```python
class LinearLayer(tf.keras.layers.Layer):
    def __init__(self):
        super().__init__()

    def build(self, input_shape):     # Here input_shape is a TensorShape.
        self.w = self.add_variable(name='w',
            shape=[input_shape[-1], 1], initializer=tf.zeros_initializer())
        self.b = self.add_variable(name='b',
            shape=[1], initializer=tf.zeros_initializer())

    def call(self, X):
        y_pred = tf.matmul(X, self.w) + self.b
        return y_pred
```

With the same way, we can call our custom layers `LinearLayer`:

```python
class Linear(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.layer = LinearLayer()

    def call(self, input):
        output = self.layer(input)
        return output
```

4.7 Graph Execution Mode *

In fact, these models above will be compatible with both Eager Execution mode and Graph Execution mode if we write them more carefully[3]. Please notice, `model(input_tensor)` is only needed to run once for building a graph under Graph Execution mode.

For example, we can also call the linear model built in the first section of this chapter and do linear regression by the following codes:

```python
    model = Linear()
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
```

[3] In addition to the RNN model implemented in this chapter, we get the length of seq_length dynamically under Eager Execution, which enables us to easily control the expanding length of RNN dynamically, which is not supported by Graph Execution. In order to reach the same effect, we need to fix the length of seq_length or use `tf.nn.dynamic_rnn` instead (Documentation).

```
X_placeholder = tf.placeholder(name='X', shape=[None, 3], dtype=tf.float32)
y_placeholder = tf.placeholder(name='y', shape=[None, 1], dtype=tf.float32)
y_pred = model(X_placeholder)
loss = tf.reduce_mean(tf.square(y_pred - y_placeholder))
train_op = optimizer.minimize(loss)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(100):
        sess.run(train_op, feed_dict={X_placeholder: X, y_placeholder: y})
    print(sess.run(model.variables))
```

CHAPTER 5

TensorFlow Extensions

This chapter introduces some of the most commonly used TensorFlow extensions. Although these features are not "must", they make the process of model training and calling more convenient.

Prerequisites:

- Python serialization module Pickle (not required)
- Python special function parameters **kwargs (not required)

5.1 Checkpoint: Saving and Restoring Variables

Usually, we hope to save the trained parameters (variables) after the model training is completed. By loading the model and parameters when you need model, you can get the trained model directly. Perhaps the first thing you think of is to store `model.variables` with the Python serialization module `pickle`. But unfortunately, TensorFlow's variable type `ResourceVariable` cannot be serialized.

Fortunately, TensorFlow provides a powerful variable saving and restoring class tf.train.Checkpoint , which can save and restore all objects in the TensorFlow containing the Checkpointable State by `save()` and `restore()` methods. Specifically, `tf.train.Optimizer` implementations, `tf.Variable`, `tf.keras.Layer` implementations or `tf.keras.Model` implementations can all be saved. Its usage is very simple, we first declare a Checkpoint:

```
checkpoint = tf.train.Checkpoint(model=model)
```

Here the initialization parameter passed to `tf.train.Checkpoint()` is special, it is a `**kwargs`. Specifically, it is a series of key-value pairs, and the keys can be taken at will, and the values are objects that need to be saved. For example, if we want to save a model instance `model` that inherits `tf.keras.Model` and an optimizer `optimizer` that inherits `tf.train.Optimizer`, we can write:

```
checkpoint = tf.train.Checkpoint(myAwesomeModel=model, myAwesomeOptimizer=optimizer)
```

Here `myAwesomeModel` is any key we take to save the model `model`. Note that we will also use this key when restoring variables.

Next, when the trained model needs to be saved, use:

```
checkpoint.save(save_path_with_prefix)
```

is fine. `save_path_with_prefix` is the directory and prefix of the saved file. For example, if you create a folder named "save" in the source directory and call `checkpoint.save('./save/model.ckpt')` once, we can find three files in the directory named `checkpoint`, `model.ckpt-1.index`, and `model.ckpt-1.data-00000-of-00001`, which record variable information. The `checkpoint.save()` method can be run multiple times. Each time we will get an .index file and a .data file. The serial number increase gradually.

When you need to reload previously saved parameters for models elsewhere, you need to instantiate a checkpoint again, while keeping the keys consistent. Then call the restore method of checkpoint. Just like this:

```
model_to_be_restored = MyModel()                                        # The same model of the␣
↪parameter to be restored
checkpoint = tf.train.Checkpoint(myAwesomeModel=model_to_be_restored)   # The key remains as
↪"myAwesomeModel"
checkpoint.restore(save_path_with_prefix_and_index)
```

Then the model variables are restored. `save_path_with_prefix_and_index` is the directory + prefix + number of the previously saved file. For example, calling `checkpoint.restore('./save/model.ckpt-1')` will load the file with the prefix `model.ckpt` and sequence number 1 to restore the model.

When saving multiple files, we often want to load the most recent one. You can use an assistant function `tf.train.latest_checkpoint(save_path)` to return the file name of the most recent checkpoint in the directory. For example, if there are 10 saved files from `model.ckpt-1.index` to `model.ckpt-10.index` in the save directory, `tf.train.latest_checkpoint('./save')` then returns `./save/model.ckpt-10` .

In general, the typical framework for restoring and saving variables is as follows:

```
# train.py - Model training phase

model = MyModel()
checkpoint = tf.train.Checkpoint(myModel=model)     # Instantiate Checkpoint, specify the save␣
↪object as model (if you need to save the optimizer's parameters, you can also add it)
```

```
# Model training code
checkpoint.save('./save/model.ckpt')                # Save the parameters to a file after the␣
↪model is trained, or save it periodically during the training process.
```

```
# test.py - Model use phase

model = MyModel()
checkpoint = tf.train.Checkpoint(myModel=model)          # Instantiate Checkpoint, specify the␣
↪recovery object as model
checkpoint.restore(tf.train.latest_checkpoint('./save'))    # Restore model parameters from file
# Model usage code
```

By the way, `tf.train.Checkpoint` is more powerful than the `tf.train.Saver`, which is commonly used in previous versions, because it supports "delayed" recovery variables under Eager Execution. Specifically, when `checkpoint.restore()` is called but the variables in the model have not yet been created, Checkpoint can wait until the variable is created before restoring the value. Under "Eager Execution" mode, the initialization of each layer in the model and the creation of variables are performed when the model is first called (the advantage is that the shape of the variable can be automatically determined based on the input tensor shape, without manual specification). This means that when the model has just been instantiated, there is actually no variable in it. At this time, using the previous method to recover the variable value will definitely cause an error. For example, you can try to save the parameters of model by calling the `save_weight()` method of `tf.keras.Model` in train.py, and call `load_weight()` method immediately after instantiating the model in test.py, it will cause an error. Only after calling the model and then run the `load_weight()` method can you get the correct result. It is obvious that `tf.train.Checkpoint` can bring us considerable convenience in this case. In addition, `tf.train.Checkpoint` also supports the Graph Execution mode.

Finally, an example is provided. The previous chapter' s multilayer perceptron model shows the preservation and loading of model variables:

```
import tensorflow as tf
import numpy as np
from zh.model.mlp.mlp import MLP
from zh.model.mlp.utils import DataLoader

tf.enable_eager_execution()
mode = 'test'
num_batches = 1000
batch_size = 50
learning_rate = 0.001
data_loader = DataLoader()


def train():
    model = MLP()
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    checkpoint = tf.train.Checkpoint(myAwesomeModel=model)      # 实例化 Checkpoint, 设置保存对象为
model
    for batch_index in range(num_batches):
        X, y = data_loader.get_batch(batch_size)
        with tf.GradientTape() as tape:
            y_logit_pred = model(tf.convert_to_tensor(X))
            loss = tf.losses.sparse_softmax_cross_entropy(labels=y, logits=y_logit_pred)
            print("batch %d: loss %f" % (batch_index, loss.numpy()))
        grads = tape.gradient(loss, model.variables)
        optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))
        if (batch_index + 1) % 100 == 0:                        # 每隔 100 个 Batch 保存一次
            checkpoint.save('./save/model.ckpt')                # 保存模型参数到文件


def test():
    model_to_be_restored = MLP()
    checkpoint = tf.train.Checkpoint(myAwesomeModel=model_to_be_restored)      # 实例化 Checkpoint,
设置恢复对象为新建立的模型 model_to_be_restored
    checkpoint.restore(tf.train.latest_checkpoint('./save'))    # 从文件恢复模型参数
    num_eval_samples = np.shape(data_loader.eval_labels)[0]
    y_pred = model_to_be_restored.predict(tf.constant(data_loader.eval_data)).numpy()
    print("test accuracy: %f" % (sum(y_pred == data_loader.eval_labels) / num_eval_samples))


if __name__ == '__main__':
    if mode == 'train':
        train()
    if mode == 'test':
        test()
```

After the save folder is created in the source directory and the model is trained, the model variable data stored every 100 batches will be stored in the save folder. Change line 7 to `model = 'test'` and run the code again. The model will be restored directly using the last saved variable value and tested on the test set. You can directly get an accuracy of about 95%.

5.2 TensorBoard: Visualization of the Training Process

Sometimes you want to see how the various parameters change during the model training (such as the value of the loss function). Although it can be viewed through the terminal output, it is sometimes not intuitional enough. TensorBoard is a tool that helps us visualize the training process.

Currently, TensorBoard support in Eager Execution mode is still in tf.contrib.summary, and there may be more changes in the future. So here are just a simple example. First, create a folder (such as ./tensorboard) in the source directory to store the TensorBoard record file, and instantiate a logger in the code:

```
summary_writer = tf.contrib.summary.create_file_writer('./tensorboard')
```

Next, put the code of training part in the context of `summary_writer.as_default()` and `tf.contrib.summary.always_record_summaries()` using "with" statement, and run `tf.contrib.summary.scalar(name, tensor, step=batch_index)` for the parameters that need to be logged (usually scalar). The "step" parameter here can be set according to your own needs, and can commonly be set to be the batch number in the current training process. The overall framework is as follows:

```
summary_writer = tf.contrib.summary.create_file_writer('./tensorboard')
with summary_writer.as_default(), tf.contrib.summary.always_record_summaries():
    # Start model training
    for batch_index in range(num_batches):
        # Training code, the current loss of batch is put into the variable "loss"
        tf.contrib.summary.scalar("loss", loss, step=batch_index)
        tf.contrib.summary.scalar("MyScalar", my_scalar, step=batch_index)  # You can also add␣
↪other variables
```

Each time you run `tf.contrib.summary.scalar()`, the logger writes a record to the log file. In addition to the simplest scalar, TensorBoard can also visualize other types of data (such as images, audio, etc.) as described in the API document.

When we want to visualize the training process, open the terminal in the source directory (and enter the TensorFlow conda environment if necessary), run:

```
tensorboard --logdir=./tensorboard
```

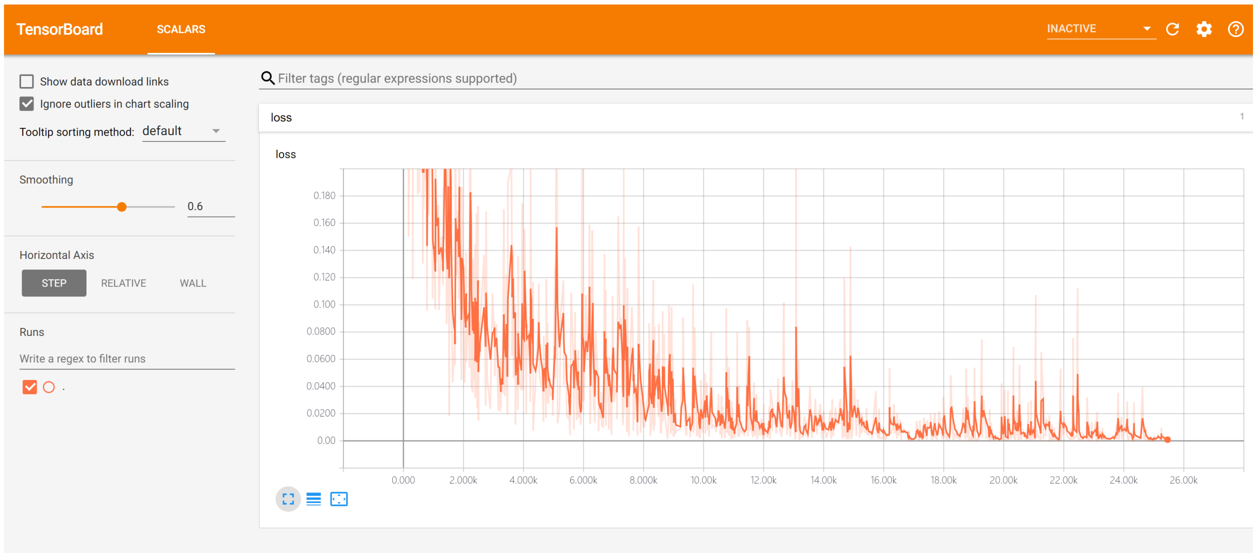
Then use the browser to visit the URL output by the terminal (usually http://computer_name:6006), you can visit the visible interface of TensorBoard, as shown below:

By default, TensorBoard updates data every 30 seconds. However, you can also manually refresh by clicking the refresh button in the upper right corner.

When using TensorBoard, please notice the following notes:

- If you want to retrain, you need to delete the information in the record folder and restart TensorBoard (or create a new record folder and open TensorBoard with the `--logdir` parameter set to be the newly created folder);
- Language of the record directory path should all be English.

Finally, we provide an example of the previous chapter' s multilayer perceptron model showing the use of TensorBoard:



Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
Horizontal Axis STEP RELATIVE WALL

```
import tensorflow as tf
import numpy as np
from zh.model.mlp.mlp import MLP
from zh.model.mlp.utils import DataLoader

tf.enable_eager_execution()
num_batches = 10000
batch_size = 50
learning_rate = 0.001
model = MLP()
data_loader = DataLoader()
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
summary_writer = tf.contrib.summary.create_file_writer('./tensorboard')     # 实例化记录器
with summary_writer.as_default(), tf.contrib.summary.always_record_summaries():
    for batch_index in range(num_batches):
        X, y = data_loader.get_batch(batch_size)
        with tf.GradientTape() as tape:
            y_logit_pred = model(tf.convert_to_tensor(X))
            loss = tf.losses.sparse_softmax_cross_entropy(labels=y, logits=y_logit_pred)
            print("batch %d: loss %f" % (batch_index, loss.numpy()))
            tf.contrib.summary.scalar("loss", loss, step=batch_index)       # 记录当前 loss
        grads = tape.gradient(loss, model.variables)
        optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))
```

5.3 GPU Usage and Allocation

Usually the scenario is: there are many students/researchers in the lab/company research group who need to use the GPU, but there is only one multi-card machine. At this time, you need to pay attention to how to allocate graphics resources.

The command `nvidia-smi` can view the existing GPU and the usage of the machine (in Windows, add `C:\Program Files\NVIDIA Corporation\NVSMI` to the environment variable "Path", or in Windows 10 you can view the graphics card information using the Performance tab of the Task Manager).

Use the environment variable `CUDA_VISIBLE_DEVICES` to control the GPU used by the program. Assume that, on a four-card machine, GPUs 0, 1 are in use and GPUs 2, 3 are idle. Then type in the Linux terminal:

```
export CUDA_VISIBLE_DEVICES=2,3
```

or add this in the code,

```
import os
os.environ['CUDA_VISIBLE_DEVICES'] = "2,3"
```

to specify that the program runs only on GPUs 2, 3.

By default, TensorFlow will use almost all of the available graphic memory to avoid performance loss caused by memory fragmentation. You can set the strategy for TensorFlow to use graphic memory through the `tf.ConfigProto` class. The specific way is to instantiate a `tf.ConfigProto` class, set the parameters, and specify the "config" parameter when running `tf.enable_eager_execution()`. The following code use the `allow_growth` option to set TensorFlow to apply for memory space only when necessary:

```
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
tf.enable_eager_execution(config=config)
```

The following code sets TensorFlow to consume 40% of GPU memory by the `per_process_gpu_memory_fraction` option:

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
tf.enable_eager_execution(config=config)
```

Under "Graph Execution", you can also pass the tf.ConfigPhoto class to set up when instantiating a new session.

CHAPTER 6

Appendix: Static TensorFlow

6.1 TensorFlow 1+1

Essentially, TensorFlow is a symbolic computational framework (based on computational graph). Here is a "Hello World" example of computing 1+1.

```
import tensorflow as tf

# 定义一个“计算图”
a = tf.constant(1)  # 定义一个常量 Tensor (张量)
b = tf.constant(1)
c = a + b  # 等价于 c = tf.add(a, b), c 是张量 a 和张量 b 通过 Add 这一 Operation (操作) 所形成的新张量

sess = tf.Session()     # 实例化一个 Session (会话)
c_ = sess.run(c)        # 通过 Session 的 run() 方法对计算图里的节点 (张量) 进行实际的计算
print(c_)
```

Output:

```
2
```

The program above is capable of computing 1+1 only, the following program, however, shows how to use TensorFlow to compute the sum of any two numbers through the parameter `feed_dict=` of `tf.placeholder()` and `sess.run()`:

```
import tensorflow as tf

a = tf.placeholder(dtype=tf.int32)  # 定义一个占位符 Tensor
b = tf.placeholder(dtype=tf.int32)
c = a + b

a_ = input("a = ")  # 从终端读入一个整数并放入变量 a_
b_ = input("b = ")

sess = tf.Session()
c_ = sess.run(c, feed_dict={a: a_, b: b_})  # feed_dict 参数传入为了计算 c 所需要的张量的值
print("a + b = %d" % c_)
```

Terminal:

```
>>> a = 2
>>> b = 3
a + b = 5
```

Variable is a special type of tensor, which is built using `tf.get_variable()`. Just like variables in common progamming language, a `Variable` should be initialized before used and its value can be modified during computation in the computational graph. The following example shows how to create a `Variable`, initialize its value to 0, and increment by one.

```
import tensorflow as tf

a = tf.get_variable(name='a', shape=[])
initializer = tf.assign(a, 0)   # tf.assign(x, y) 返回一个“将张量 y 的值赋给变量 x”的操作
a_plus_1 = a + 1    # 等价于 a + tf.constant(1)
plus_one_op = tf.assign(a, a_plus_1)

sess = tf.Session()
sess.run(initializer)
for i in range(5):
    sess.run(plus_one_op)                  # 对变量 a 执行加一操作
    a_ = sess.run(a)                       # 获得变量 a 的值并存入 a_
    print(a_)
```

Output:

```
1.0
2.0
3.0
4.0
5.0
```

The following code is equivalent to the code shown above. It specifies the initializer upon declaring variables and initializes all variables at once by `tf.global_variables_initializer()`, which is used more often in practical projects:

```
import tensorflow as tf

a = tf.get_variable(name='a', shape=[], initializer=tf.zeros_initializer)   # 指定初始化器为全 0 初始化
a_plus_1 = a + 1
plus_one_op = tf.assign(a, a_plus_1)

sess = tf.Session()
sess.run(tf.global_variables_initializer()) # 初始化所有变量
for i in range(5):
    sess.run(plus_one_op)
    a_ = sess.run(a)
    print(a_)
```

Matrix and tensor calculation is the basic operation in scientific computation (including Machine Learning). The program shown below is to demonstrate how to calculate the product of the two matrices $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ and $\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$:

```
import tensorflow as tf

A = tf.ones(shape=[2, 3])   # tf.ones(shape) 定义了一个形状为 shape 的全 1 矩阵
B = tf.ones(shape=[3, 2])
C = tf.matmul(A, B)

sess = tf.Session()
C_ = sess.run(C)
print(C_)
```

Output:

```
[[3. 3.]
 [3. 3.]]
```

Placeholders and Variables are also allowed to be vector, matrix and even higher dimentional tensor.

6.2 A Basic Example: Linear Regression

Unlike previous NumPy and Eager Execution mode, TensorFlow’ s Graph Execution mode uses **symbolic programming** for numerical operations. First, we need to abstract the computational processes into a Dataflow Graph, and represent the inputs, operations and outputs with symbolized nodes. Then, we continually send the data to the input nodes, let the data be calculated and flow along the dataflow graph, and finally reach the specific output nodes we want. The following code shows how to accomplish the same task as the code does in previous section based on TensorFlow’ s symbolic programming approach, where `tf.placeholder()` can be regarded as a kind of “symbolic input node” , using `tf.get_variable()` to define the parameters of the model (the tensor of the Variable type can be assigned using `tf.assign()`), and `sess.run(output_node, feed_dict={input_node: data})` can be thought of as a process which sends data to the input node, calculates along the dataflow graph and reach the output node and eventually return values.

```python
import tensorflow as tf

# 定义数据流图
learning_rate_ = tf.placeholder(dtype=tf.float32)
X_ = tf.placeholder(dtype=tf.float32, shape=[5])
y_ = tf.placeholder(dtype=tf.float32, shape=[5])
a = tf.get_variable('a', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)
b = tf.get_variable('b', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)

y_pred = a * X_ + b
loss = tf.constant(0.5) * tf.reduce_sum(tf.square(y_pred - y_))

# 反向传播，手动计算变量（模型参数）的梯度
grad_a = tf.reduce_sum((y_pred - y_) * X_)
grad_b = tf.reduce_sum(y_pred - y_)

# 梯度下降法，手动更新参数
new_a = a - learning_rate_ * grad_a
new_b = b - learning_rate_ * grad_b
update_a = tf.assign(a, new_a)
update_b = tf.assign(b, new_b)

train_op = [update_a, update_b]
# 数据流图定义到此结束
# 注意，直到目前，我们都没有进行任何实质的数据计算，仅仅是定义了一个数据图

num_epoch = 10000
learning_rate = 1e-3
with tf.Session() as sess:
    # 初始化变量 a 和 b
    tf.global_variables_initializer().run()
    # 循环将数据送入上面建立的数据流图中进行计算和更新变量
    for e in range(num_epoch):
        sess.run(train_op, feed_dict={X_: X, y_: y, learning_rate_: learning_rate})
    print(sess.run([a, b]))
```

In the two examples above, we manually calculated the partial derivatives of the loss function with regard to each parameter. But when both the model and the loss function become very complicated (especially in deep learning models), the workload of manual derivation is unacceptable. TensorFlow provides an **automatic derivation mechanism** that eliminates the hassle of manually calculating derivatives, using TensorFlow' s derivation function `tf.gradients(ys, xs)` to compute the partial derivatives of the loss function with regard to a and b. Thus, the two lines of code in the previous section for calculating derivatives manually,

```
# 反向传播，手动计算变量（模型参数）的梯度
grad_a = tf.reduce_sum((y_pred - y_) * X_)
grad_b = tf.reduce_sum(y_pred - y_)
```

could be replaced by

```
grad_a, grad_b = tf.gradients(loss, [a, b])
```

and the result won' t change.

Moreover，TensorFlow has many kinds of **optimizer**, which can complete derivation and gradient update together at the same time. The code in the previous section,

```
# 反向传播，手动计算变量（模型参数）的梯度
grad_a = tf.reduce_sum((y_pred - y_) * X_)
grad_b = tf.reduce_sum(y_pred - y_)

# 梯度下降法，手动更新参数
new_a = a - learning_rate_ * grad_a
new_b = b - learning_rate_ * grad_b
update_a = tf.assign(a, new_a)
update_b = tf.assign(b, new_b)

train_op = [update_a, update_b]
```

could be replaced by

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate_)
grad = optimizer.compute_gradients(loss)
train_op = optimizer.apply_gradients(grad)
```

Here, we first instantiate a gradient descent optimizer `tf.train.GradientDescentOptimizer()` in Tensor-

Flow and set the learning rate. Then use its `compute_gradients(loss)` method to find the gradients of `loss` with regard to all variables (parameters). Finally, through the method `apply_gradients(grad)`, the variables (parameters) are updated according to the previously calculated gradients.

These three lines of code are equivalent to the following line of code:

```
train_op = tf.train.GradientDescentOptimizer(learning_rate=learning_rate_).minimize(loss)
```

The simplified code is as follows:

```
import tensorflow as tf

learning_rate_ = tf.placeholder(dtype=tf.float32)
X_ = tf.placeholder(dtype=tf.float32, shape=[5])
y_ = tf.placeholder(dtype=tf.float32, shape=[5])
a = tf.get_variable('a', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)
b = tf.get_variable('b', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)

y_pred = a * X_ + b
loss = tf.constant(0.5) * tf.reduce_sum(tf.square(y_pred - y_))

# 反向传播，利用 TensorFlow 的梯度下降优化器自动计算并更新变量（模型参数）的梯度
train_op = tf.train.GradientDescentOptimizer(learning_rate=learning_rate_).minimize(loss)

num_epoch = 10000
learning_rate = 1e-3
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    for e in range(num_epoch):
        sess.run(train_op, feed_dict={X_: X, y_: y, learning_rate_: learning_rate})
    print(sess.run([a, b]))
```

Bibliography

[LeCun1998] 25. LeCun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition.” Proceedings of the IEEE, 86(11):2278-2324, November 1998. http://yann.lecun.com/exdb/mnist/

[Graves2013] Graves, Alex. “Generating Sequences With Recurrent Neural Networks.” ArXiv:1308.0850 [Cs], August 4, 2013. http://arxiv.org/abs/1308.0850.

[Mnih2013] Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. “Playing Atari with Deep Reinforcement Learning.” ArXiv:1312.5602 [Cs], December 19, 2013. http://arxiv.org/abs/1312.5602.