国防科技大学 National University of Defense Technology

深度学习工具及实践

并行与分布处理重点实验室
Science and Technology on Parallel and Distributed Laboratory(PDL)
计算机学院，国防科技大学
College of Computer, National University of Defense Technology

2020 年 3 月 2 日

国防科技大学 National University of Defense Technology

目录

国防科技大学 National University of Defense Technology

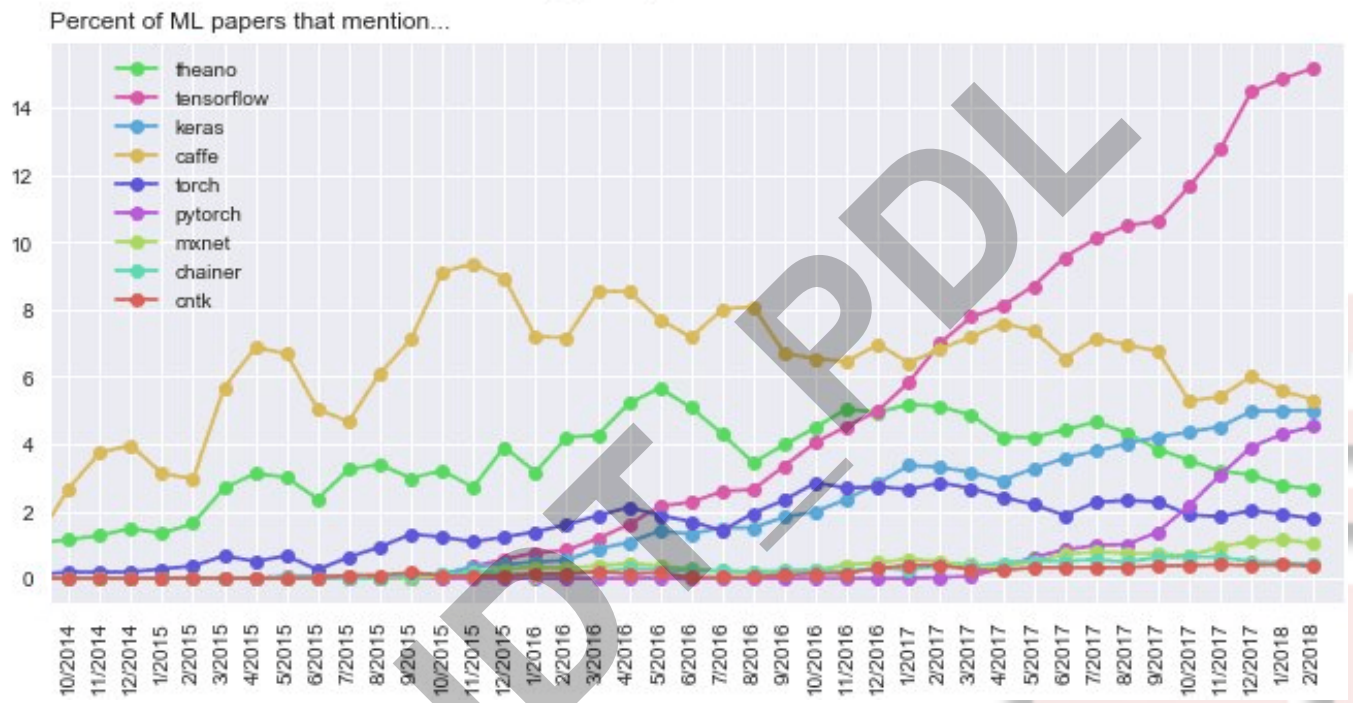
目录

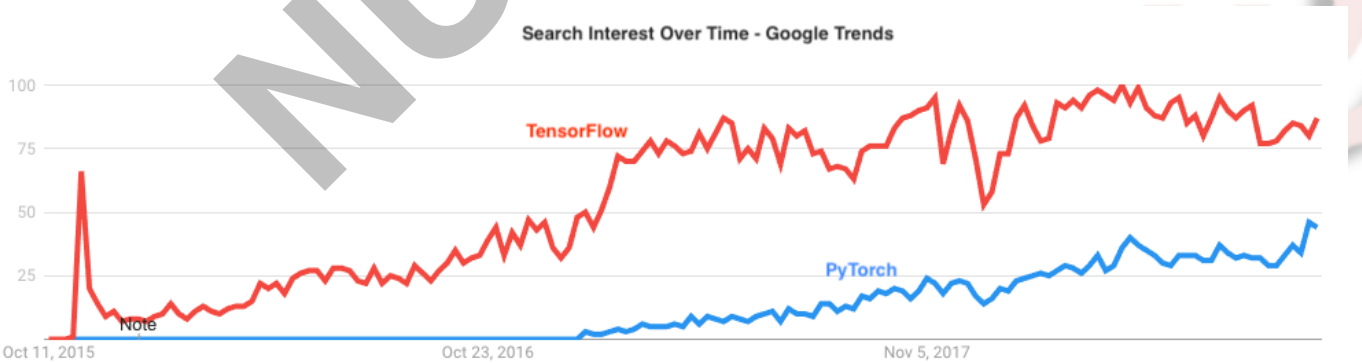
深度学习工具包

- 蛮荒时期：手写深度学习程序
  - 编程难度大
  - 对非科研人员不友好
  - 标准无法统一，难以移植和复现
- Caffe, torch, matconvnet ...
- 目前的主流：TensorFlow, Caffe 和 pyTorch

国防科技大学 National University of Defense Technology

深度学习工具包





深度学习工具包

- 为什么要用 TensorFlow? PyTorch 不香吗?
  - 自 TensorFlow 2.0 开始，易用度追上了 PyTorch
  - TensorFlow：谷歌；PyTorch：FaceBook (10000 亿 v.s. 6324 亿)
  - 谷歌具有强大的科研实力，会有更多先进算法基于 TensorFlow
  - 工业上 TensorFlow 更成熟

TensorFlow 和 Python

- 简单理解：TensorFlow(TF) 就是 Python 中调用的一个库
- 怎样熟练 TF 库? 和其他库一样
  - 数据结构：TF 定义数据如何和 Python 中其他数据进行交互
  - 算法思想：TF 进行计算的编程思想、编程模型
  - 熟练 TF 库中常用的函数、工具

TensorFlow 简介

- 开源的、端到端的机器学习平台
- Google brain 开发用来做机器学习、深度学习研究的工具
- 多平台：支持 CPU/GPU，服务器、个人电脑、移动设备
- 分布并行：方便多 GPU，分布式训练
- 可扩展：支持 OP 扩展，kernal 扩展

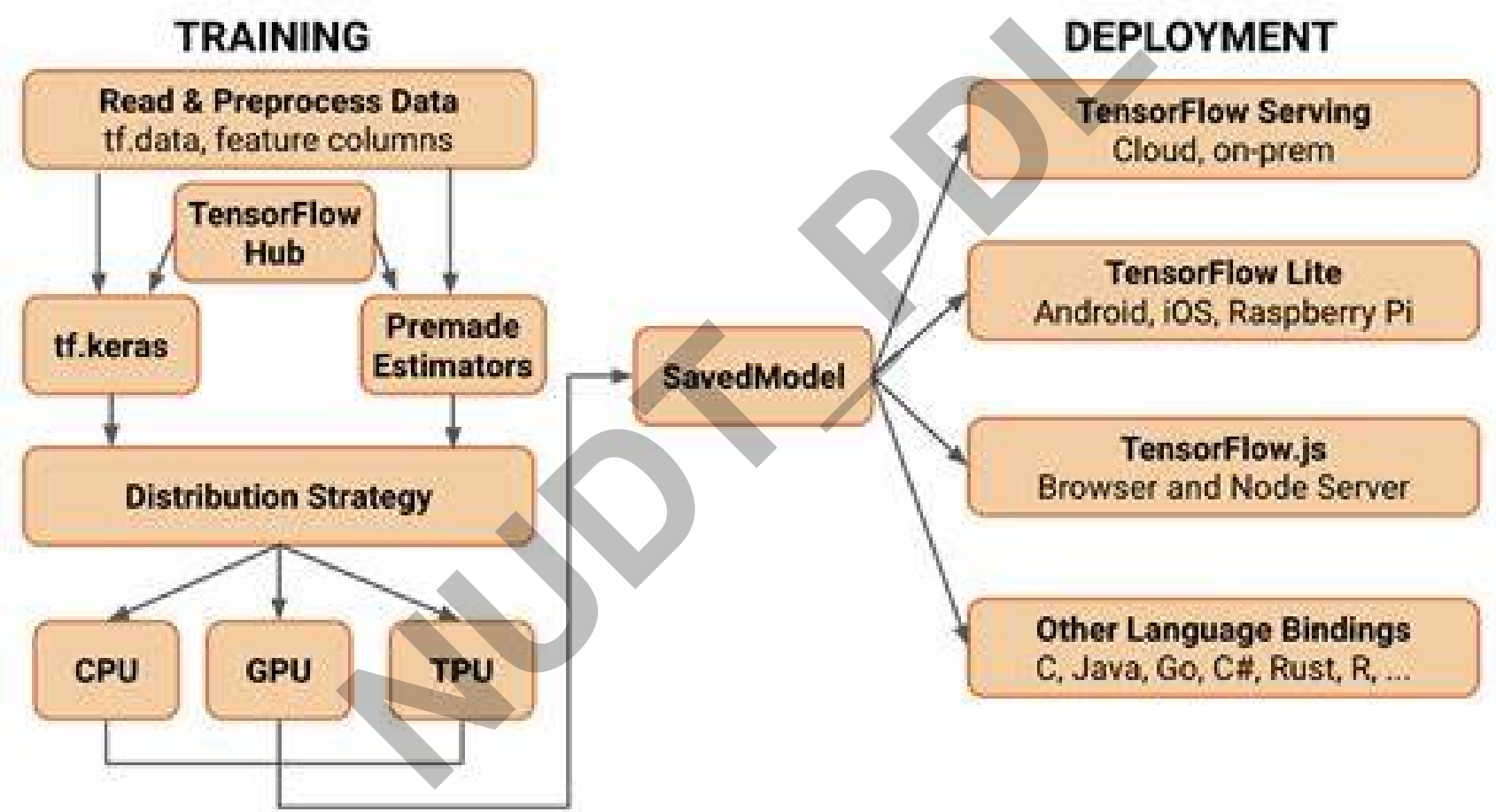
TensorFlow 简介

- 接口丰富：支持 python, java 以及 c++ 接口
- 可视化：使用 TensorBoard 可视化计算图
- 易用性：相比 Caffe 等易于学习掌握，文档资料丰富
- 社区支持：开源项目支持最多的几种框架：Tensorflow、Caffe、pyTorch

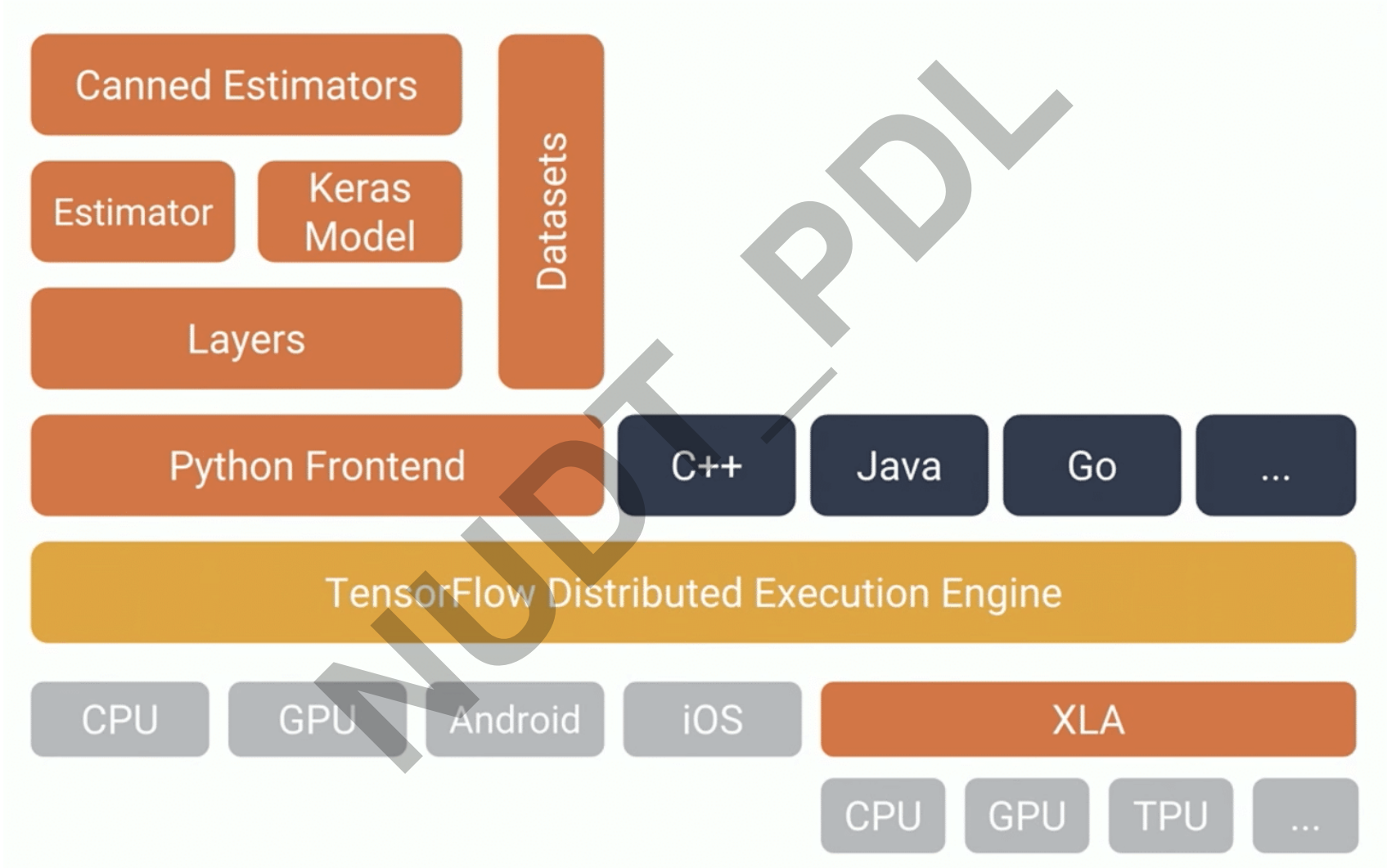
TensorFlow 易于学习

- Python 接口：适于非计算机专业学习编程
- 安装方便: Anaconda 或 pip install
- 多平台：windows, linux, mac OS
- 模型设计方便：
  - 1. 支持多种深度学习网络层
  - 2. 易于自定义层
  - 3. TensorFlow 2.0 限定：支持动态图模型，编程难度大大降低
- 多 GPU, 分布式训练支持方便

TensorFlow 基本架构



TensorFlow 层级



国防科技大学 National University of Defense Technology

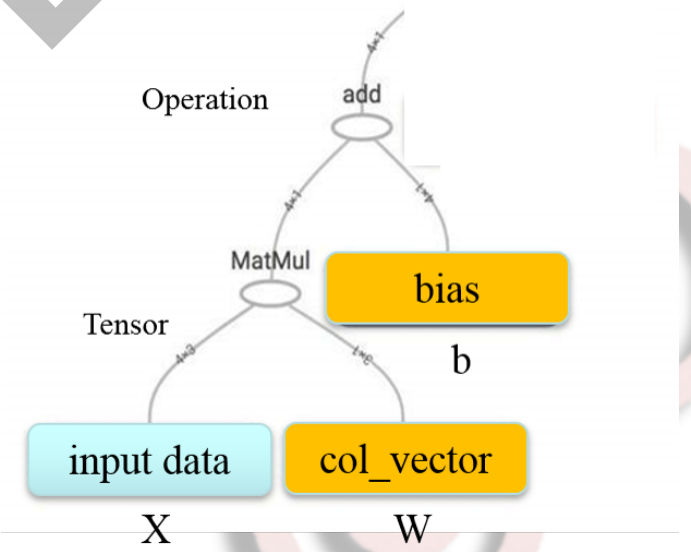
目录

TensorFlow 2.x 基本编程框架

- 计算图
- Eager Execution 机制
- 静态图与动态图
- TensorFlow 2.x 基本编程流程
  - 数据准备：Numpy, tf.data
  - 模型构建：tf.keras
  - 训练：eager execution
  - 模型保存：savedmodel

计算图

- 用于描述模型计算的过程，在 TensorFlow 编程体系中占有重要地位
- 假设要计算线性模型 $y = Wx + b$，则需要用计算图描述这一模型
- 计算图分为静态图与动态图



Eager Execution

TensorFlow 1.x: 传统执行模式

- 先画好计算图
- 再利用会话 run 计算图
- 执行过程中看不到中间结果，只能得到预定义的 output
- 不利于调试，主要与静态图机制结合

TensorFlow 2.0: Eager Execution

- 依次执行程序中的语句
- 可以看到程序的中间结果
- 方便调试，同时方便实现动态图架构

传统模式 v.s. Eager Execution

```
import tensorflow as tf  # TensorFlow 1.x 版本
v1 = tf.Variable(tf.random_uniform([3]))
v2 = tf.Variable(tf.random_uniform([3]))
sumV12 = tf.add(v1, v2)
print(v1, '\n')
print(v2, '\n')
print(sumV12, '\n')
#运行结果如下:
'''
<tf.Variable 'Variable:0' shape=(3,) dtype=float32_ref>
<tf.Variable 'Variable_1:0' shape=(3,) dtype=
    float32_ref>
Tensor("Add:0", shape=(3,), dtype=float32)
'''
```

传统模式 v.s. Eager Execution

```
import tensorflow as tf  # TensorFlow 1.x 版本

v1 = tf.Variable(tf.random_uniform([3]))
v2 = tf.Variable(tf.random_uniform([3]))
sumV12 = tf.add(v1, v2)

with tf.Session() as sess:
sess.run(tf.initialize_all_variables())
print(sess.run(v1))
print(sess.run(v2))
print(sess.run(sumV12))
#运行结果如下:
'''
[[0.2113781  0.54565406 0.9065633 ]]
[[0.37743175  0.6158673  0.43864775]]
[[0.58880985  1.1615304  1.345211  ]]
'''
```

传统模式 v.s. Eager Execution

```
import tensorflow as tf  # TensorFlow 2.x 版本

v1 = tf.random.normal([3])
v2 = tf.random.normal([3])
sumV12 = tf.add(v1, v2)
print(v1, '\n')
print(v2, '\n')
print(sumV12, '\n')
#运行结果如下:
'''
tf.Tensor([ 0.22552335 -0.7973353  -0.7176047 ], shape
    =(3,), dtype=float32)
tf.Tensor([-0.69635415  0.4780235  -1.7286636 ], shape
    =(3,), dtype=float32)
tf.Tensor([-0.4708308  -0.31931183 -2.4462683 ], shape
    =(3,), dtype=float32)
'''
```

静态图与动态图

TensorFlow 1.x: 静态图框架:

- 预先定义计算图，运行时反复使用，不能改变
- 代表：TensorFlow 1.x, Caffe
- 优点：速度更快，适合大规模部署，适合嵌入式平台

TensorFlow 2.x: 动态图框架:

- 每次运行时都会重新构建计算图，因此可以在学习过程中对计算图进行修改
- 代表：TensorFlow 2.x, PyTorch
- 优点：灵活性高，便于 debug，学习成本更低

静态图与动态图特点对比详述

静态图:

- 静态图框架设计好了不能够修改，图定义与计算分离
- 定义静态图时需要使用新的特殊语法：无法使用 if、while、for-loop 等结构，而需要特殊的由框架专门设计的语法
- 在构建图时，需要考虑到所有的情况 (即各个 if 分支图结构必须全部在图中，即使不一定会在每一次运行时使用到)，使得静态图可能非常庞大

动态图:

- 兼容 python 的各种逻辑控制语法
- 最终创建的图取决于每次运行时的条件分支选择

在静态图中使用 Python 控制流

```
a = tf.constant(12) # 使用TensorFlow 1.x库
while not tf.equal(a, 1):
    if tf.equal(a%2, 0):
        a = a / 2
    else:
        a = 3 * a + 1
    print(a)
#运行结果如下:
'''
File "tensorflow_demo.py", line 55, in <module>
   while not tf.equal(a, 1):
File "D:\xxx\ops.py", line 690, in __bool__
   raise TypeError("Using a `tf.Tensor` as a Python `
       bool` is not allowed. "
TypeError: Using a `tf.Tensor` as a Python `bool` is
    not allowed. Use `if t is not None:` instead of `if
    t:` to test if a tensor is defined, and use
    TensorFlow ops such as tf.cond to execute subgraphs
    conditioned on the value of a tensor.
```

在静态图中使用 Python 控制流

```
a = tf.constant(12.0)
def cond(a):
    return tf.not_equal(a, 1)
def body(a):
    a = tf.cond(tf.equal(tf.mod(a, 2),0), lambda: tf.
        divide(a, 2), lambda: tf.add(tf.multiply(a, 3),
        1))
    return a
b = tf.while_loop(cond, body, [a])
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    res = sess.run(b)
    print(res)
#运行结果如下:
'''
1.0
'''
```

在动态图中使用 Python 控制流

```
a = tf.constant(12) # a为tensor，因此所有相关运算全部被
    提升为tensor运算，输出为tensor
while not tf.equal(a, 1):
    if tf.equal(a%2, 0): # 运行过程中计算图随着分支改变
        a = a / 2
    else:
        a = 3 * a + 1
    print(a)
#运行结果如下:
'''
tf.Tensor(6.0, shape=(), dtype=float64)
tf.Tensor(3.0, shape=(), dtype=float64)
tf.Tensor(10.0, shape=(), dtype=float64)
tf.Tensor(5.0, shape=(), dtype=float64)
tf.Tensor(16.0, shape=(), dtype=float64)
tf.Tensor(8.0, shape=(), dtype=float64)
tf.Tensor(4.0, shape=(), dtype=float64)
tf.Tensor(2.0, shape=(), dtype=float64)
tf.Tensor(1.0, shape=(), dtype=float64)
```

TensorFlow 1.x 时代的痛苦

- TensorFlow 没有其他 python 机器学习库当中预定义的各种模型函数，如回归、神经网络等

TensorFlow 1.x 时代的痛苦

- TensorFlow 没有其他 python 机器学习库当中预定义的各种模型函数，如回归、神经网络等
- 不能直接将数据以参数形式送入

TensorFlow 1.x 时代的痛苦

- TensorFlow 没有其他 python 机器学习库当中预定义的各种模型函数，如回归、神经网络等
- 不能直接将数据以参数形式送入
- 必须手写计算图

TensorFlow 1.x 时代的痛苦

- TensorFlow 没有其他 python 机器学习库当中预定义的各种模型函数，如回归、神经网络等
- 不能直接将数据以参数形式送入
- 必须手写计算图
- 计算图不参与计算

TensorFlow 1.x 时代的痛苦

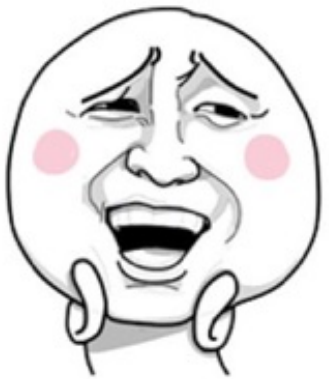
- TensorFlow 没有其他 python 机器学习库当中预定义的各种模型函数，如回归、神经网络等
- 不能直接将数据以参数形式送入
- 必须手写计算图
- 计算图不参与计算
- Session 执行计算图

TensorFlow 1.x 时代的痛苦

- TensorFlow 没有其他 python 机器学习库当中预定义的各种模型函数，如回归、神经网络等
- 不能直接将数据以参数形式送入
- 必须手写计算图
- 计算图不参与计算
- Session 执行计算图

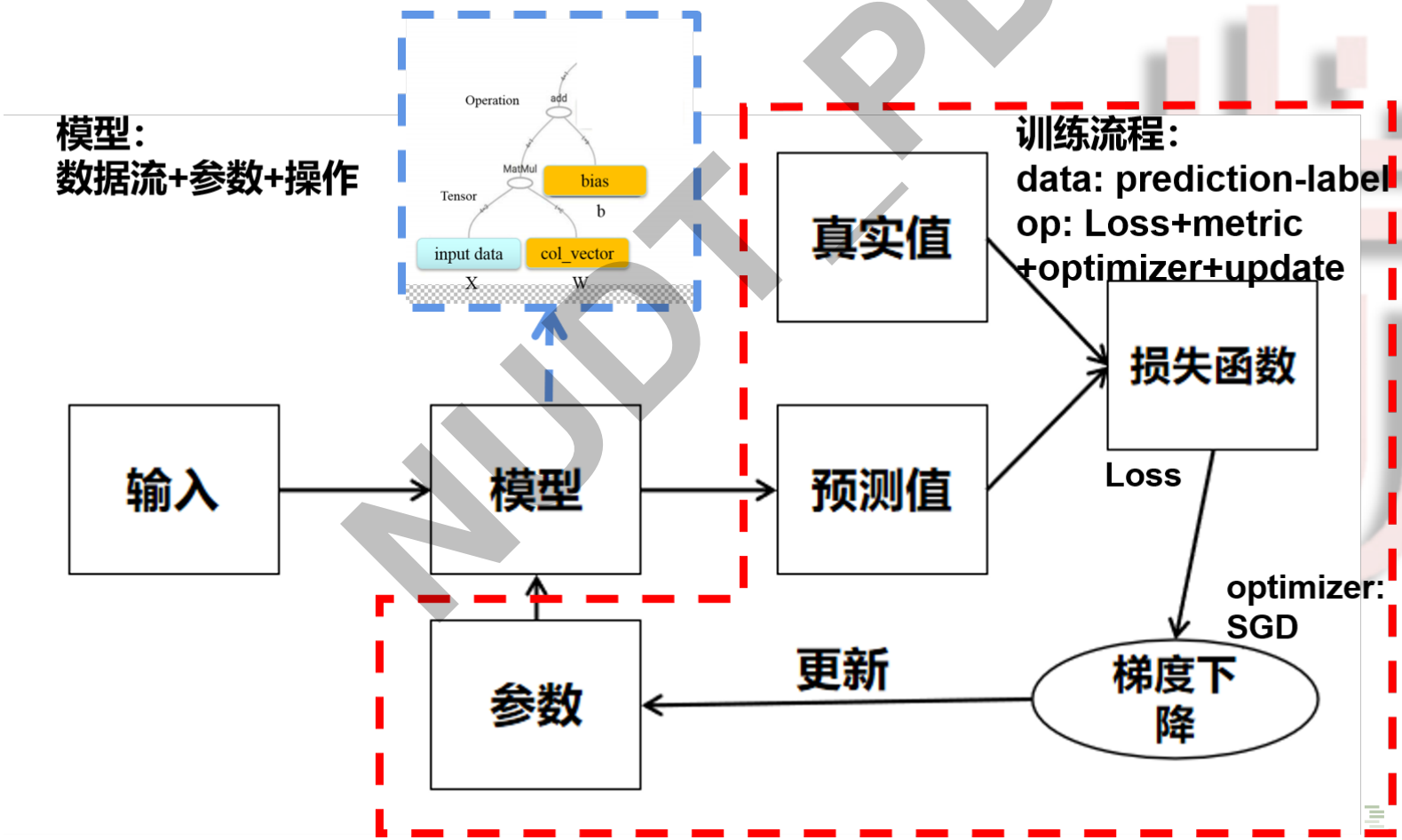
TensorFlow 2.x 时代的幸福

- 仍然没有其他 python 机器学习库当中预定义的各种模型函数，如回归、神经网络等
- 转向动态图机制
- 可以直接以参数形式传入数据
- 构建模型更加简单（Keras）
- 无需使用 Session，直接执行



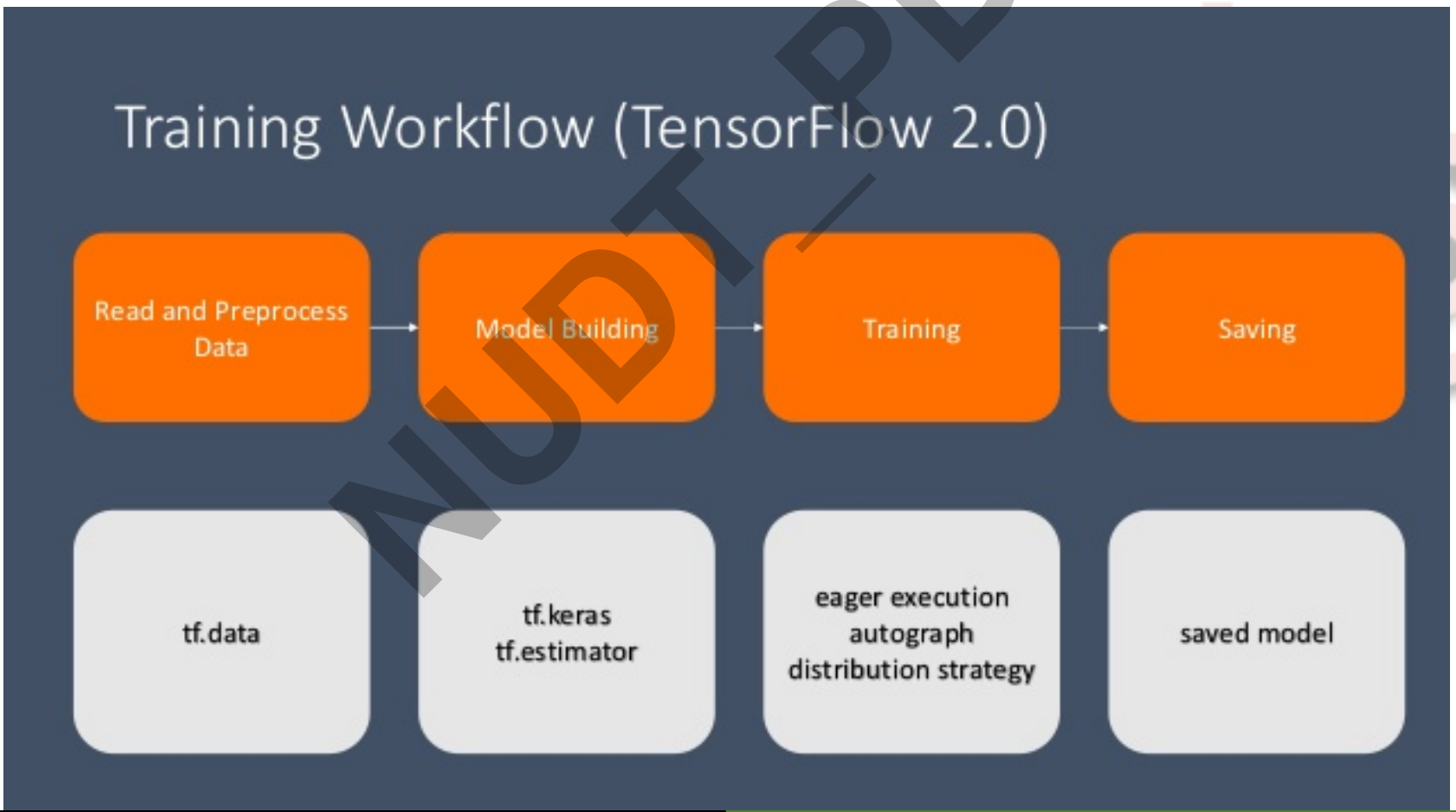
机器学习基本流程

- 准备输入数据
- 定义计算模型
- 设计损失函数、训练并调节模型参数
- 保存模型

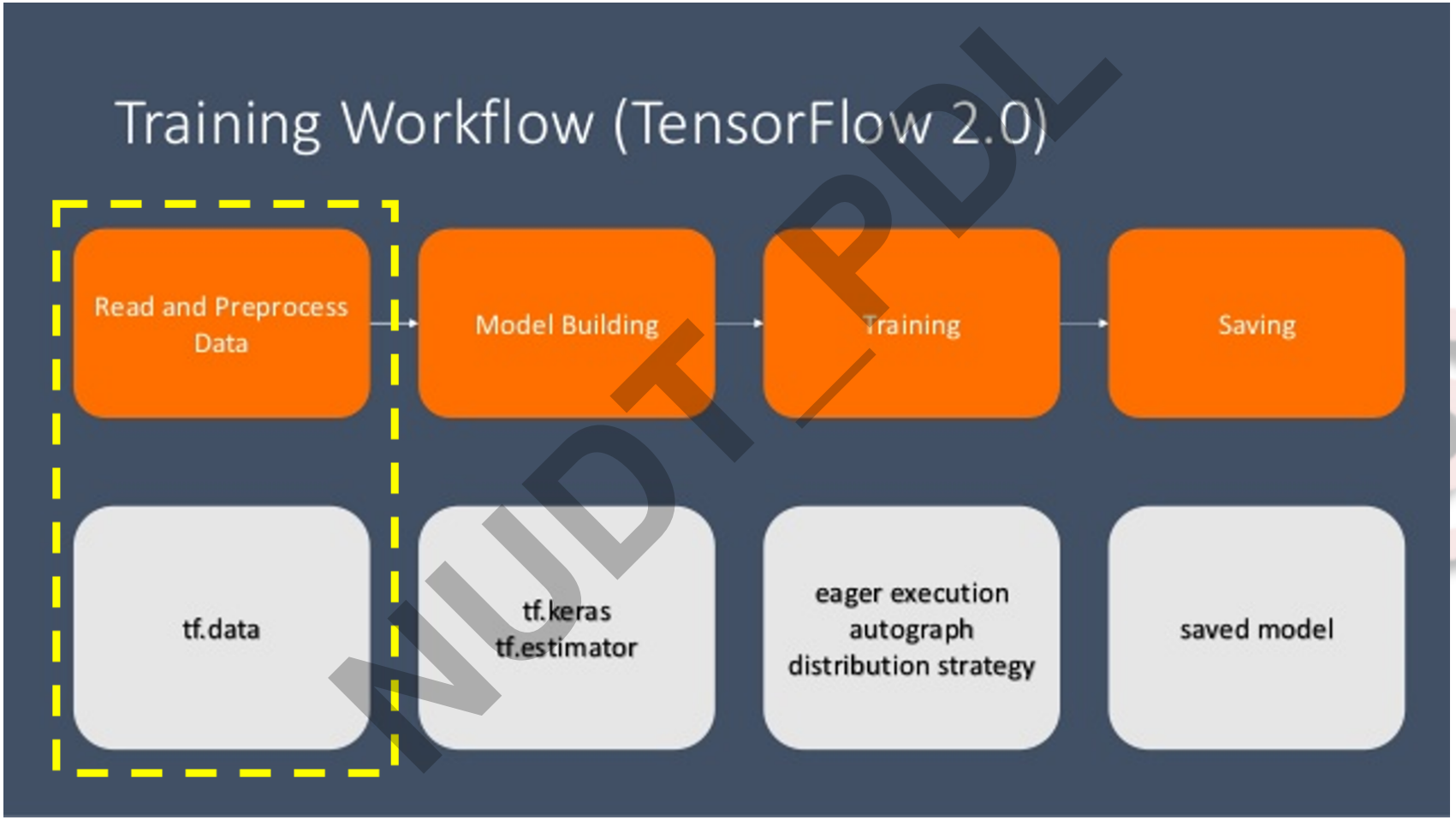


TensorFlow 2.x 基本编程流程

- 数据准备：Numpy, tf.data
- 模型构建：tf.keras
- 训练：eager execution
- 模型保存：savedmodel

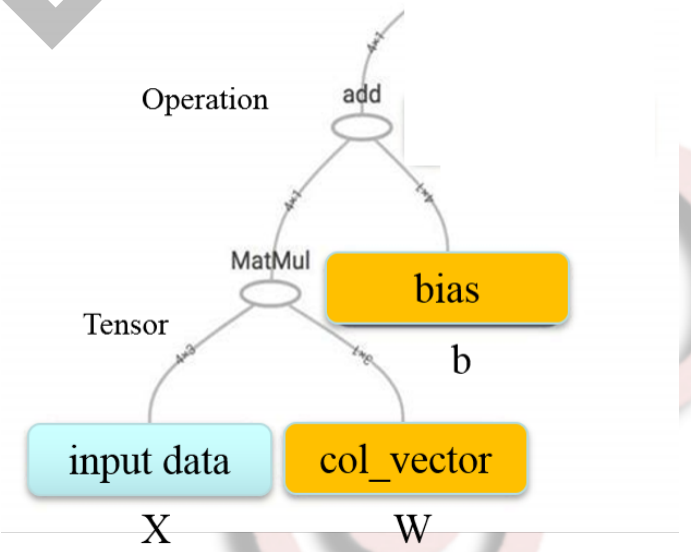


数据准备



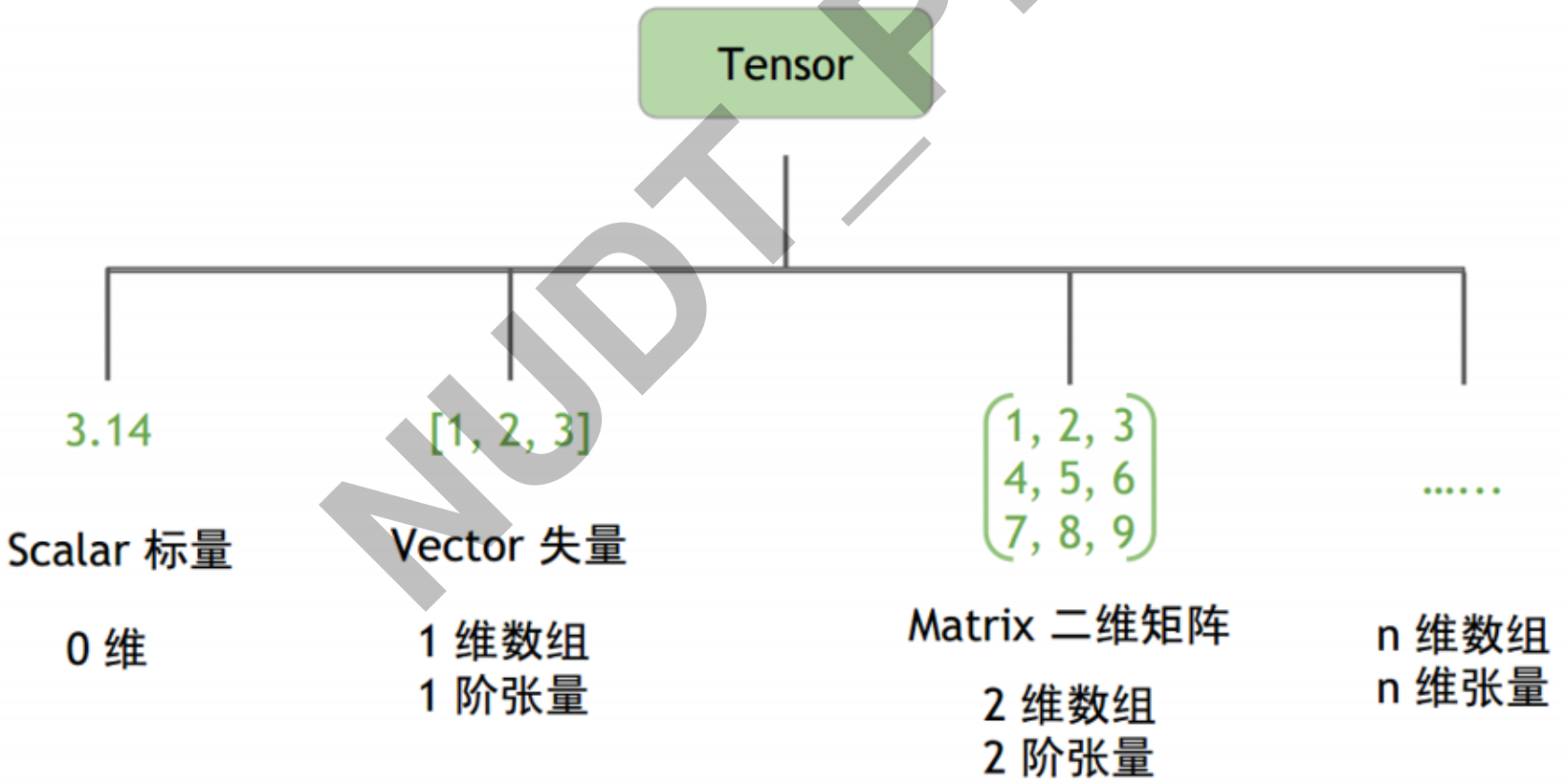
数据准备在准备什么?

- 线性模型 y = Wx + b
- Tensor: 数据传递的媒介，包括模型的输入输出及中间结果
- Variable: 变量，模型参数的表达，训练中不断改变



Tensor：TensorFlow 的基本运算单位

- TensorFlow = Tensor + Flow
  - 0 维张量: 标量（数）
  - 1 维张量: 向量
  - 2 维向量：矩阵
  - n 维向量.....



TensorFlow 2.x 张量示例

```
import tensorflow as tf

v1 = tf.random.uniform(shape=())
v2 = tf.zeros(shape=(2))

v3 = tf.constant([[1., 2.], [3., 4.]])
v4 = tf.constant([[5., 6.], [7., 8.]])

print(v1, '\n')
print(v2, '\n')
print(v3, '\n')
#运行结果如下:
'''
tf.Tensor(0.8748951, shape=(), dtype=float32)
tf.Tensor([0. 0.], shape=(2,), dtype=float32)
tf.Tensor(
[[1. 2.]
[3. 4.]], shape=(2, 2), dtype=float32)
'''
```

TensorFlow 2.x 张量计算

```
import tensorflow as tf

v3pv4 = tf.add(v3, v4)      # 计算矩阵和
v3mv4 = tf.matmul(v3, v4)   # 计算矩阵乘积

print(v3pv4, '\n')
print(v3mv4, '\n')
#运行结果如下:
'''
tf.Tensor(
[[ 6.  8.]
[10. 12.]], shape=(2, 2), dtype=float32)

tf.Tensor(
[[19. 22.]
[43. 50.]], shape=(2, 2), dtype=float32)
'''
```

数据准备

- 把计算图的输入数据准备好以便进行计算
  - tensor 与其他数据结构的关系：都是基于 numpy，有转换工具
  - 可以直接加载 Numpy 数据并送入计算图
  - 如果有大的数据集，可以使用用 TensorFlow 的 dataset 工具帮助加载 (tf.data)

数据准备

- 利用 tf.data 加载 numpy 数据

```
import tensorflow as tf
import numpy as np

X = np.array([2013, 2014, 2015])
Y = np.array([12000, 14000, 15000])

dataset = tf.data.Dataset.from_tensor_slices((X, Y))
for x, y in dataset:
    print(x, y) # x.numpy(), y.numpy()
#运行结果如下:
'''
tf.Tensor(2013, shape=(), dtype=int32) tf.Tensor(12000,
    shape=(), dtype=int32)
tf.Tensor(2014, shape=(), dtype=int32) tf.Tensor(14000,
    shape=(), dtype=int32)
tf.Tensor(2015, shape=(), dtype=int32) tf.Tensor(15000,
    shape=(), dtype=int32)
'''
```

数据准备

- 利用 tf.data 直接获得预先准备的经典数据集 (例如 MNIST)

```
import tensorflow as tf

(train_data, train_label), (_, _) = tf.keras.datasets.
    mnist.load_data()
train_data = np.expand_dims(train_data.astype(np.
    float32) / 255.0, axis=-1)      # [60000, 28, 28, 1]

mnist_dataset = tf.data.Dataset.from_tensor_slices((
    train_data, train_label))
```

利用 tf.data 进行数据预处理

- tf.data.Dataset.map(f): 对数据集中的每个元素应用函数 f, 得到一个新的数据集

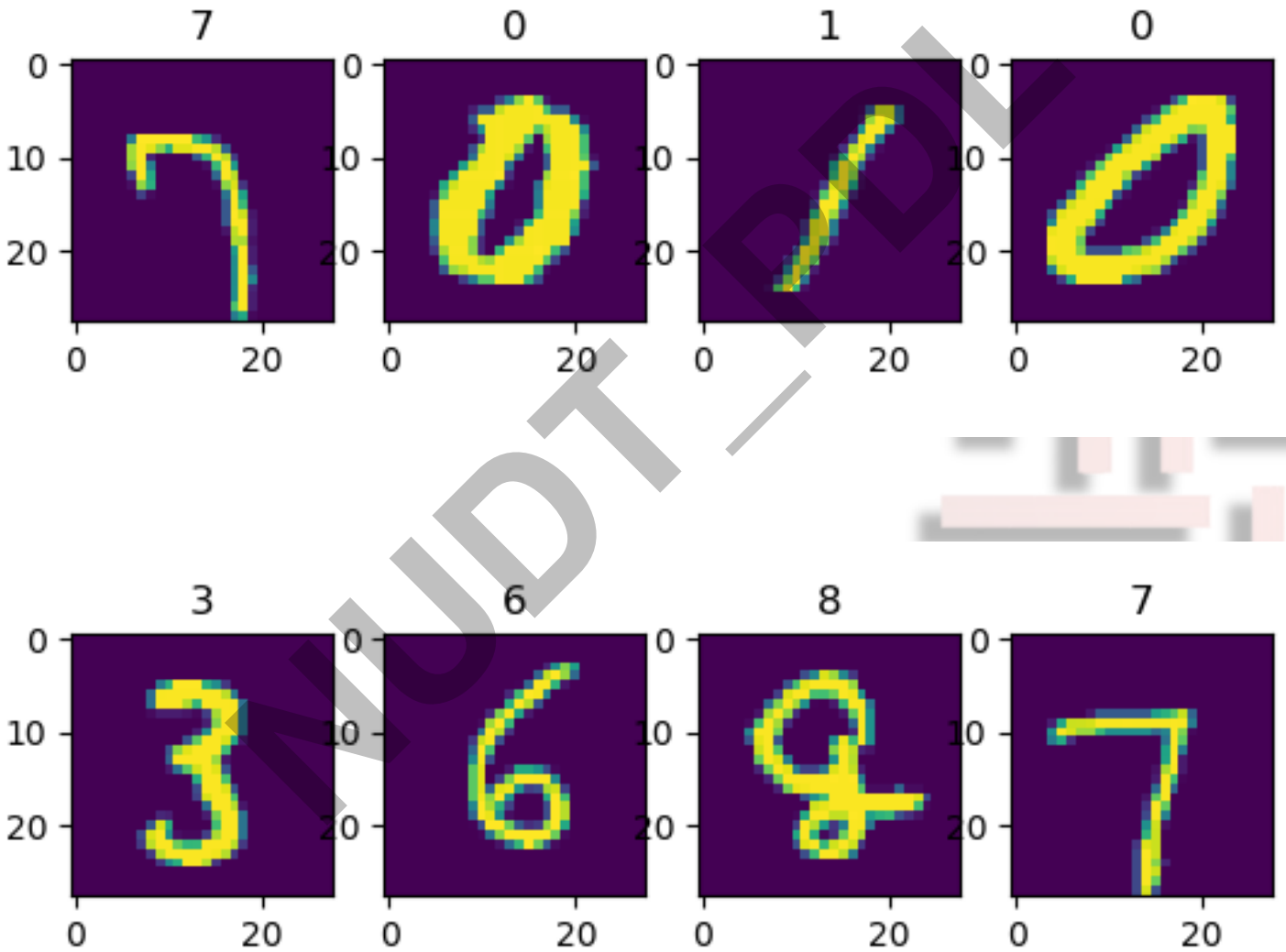
```
def rot90(image, label):
    image = tf.image.rot90(image)
    return image, label

mnist_dataset = mnist_dataset.map(rot90)
```

利用 tf.data 进行数据预处理

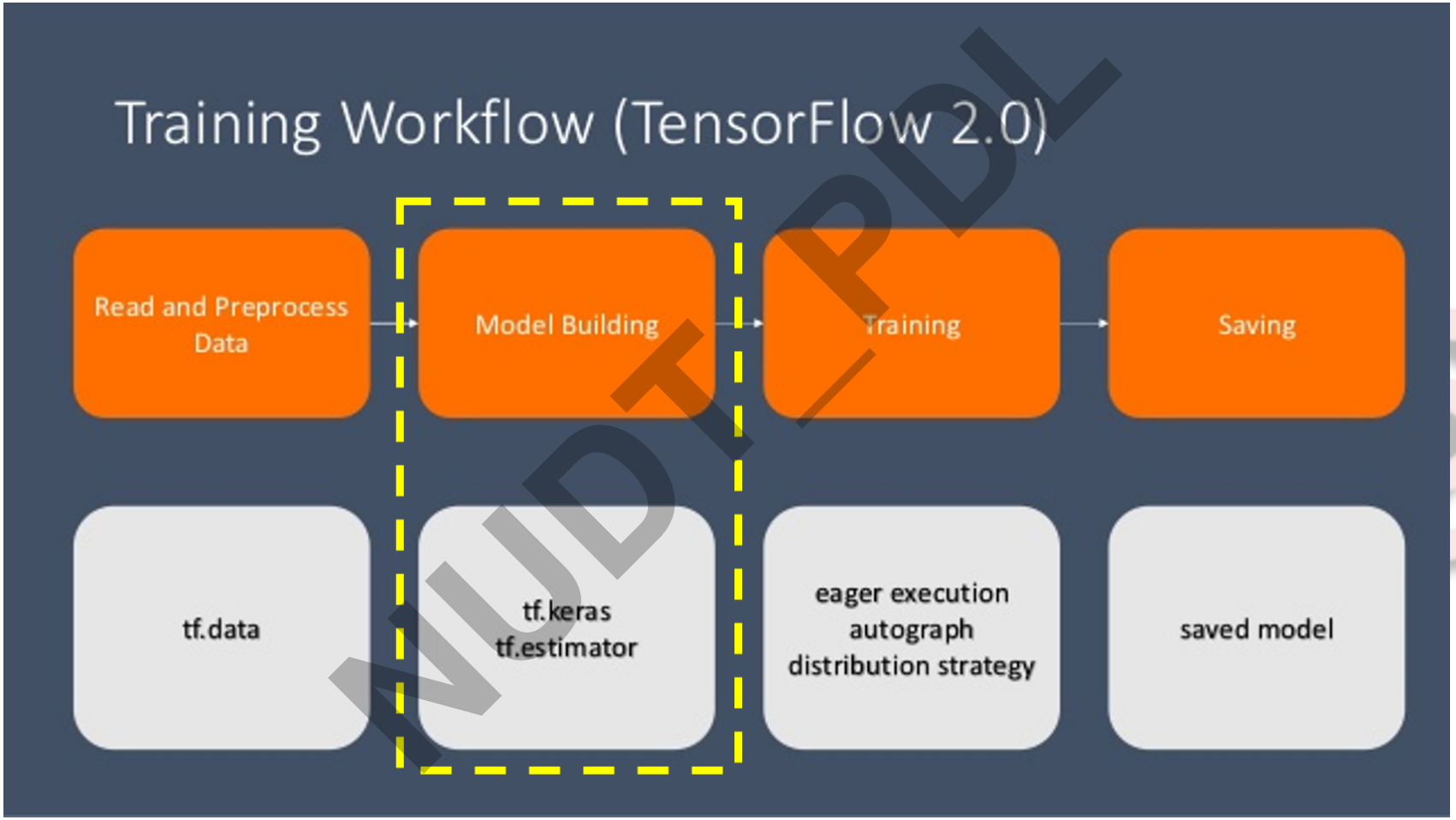
- tf.data.Dataset.shuffle(bufferSize)：将数据集打乱 (设定一个固定大小的缓冲区，并从缓冲区中随机采样，采样后的数据用后续数据替换)
- tf.data.Dataset.batch(batchSize)： 将数据集分成批次

```
mnist_dataset = mnist_dataset.shuffle(buffer_size
    =10000).batch(4)
for images, labels in mnist_dataset:
    fig, axs = plt.subplots(1, 4)
    for i in range(4):
        axs[i].set_title(labels.numpy()[i])
        axs[i].imshow(images.numpy()[i, :, :, 0])
    plt.show()
```

利用 tf.data 进行数据预处理



模型构建



TensorFlow 2.x 模型构建的几种方式

构建模型: 即画计算图，tf 提供了多层次绘图方式

- 低阶 API 实现：tf.mul(x, b)
  - 利用 Eager Execution 机制
  - 一个一个绘制计算图中的节点
  - 简单直观，但是对复杂模型力不从心
- 使用高阶 Keras 接口：model.add = keras...
  - 支持更复杂的网络结构
  - 利用预定义的砖块拼图
  - 代码更加简洁优雅
- 定义模型类: model = linear()
  - 支持自定义层
  - 更大规模拼图，抽象化及打包复用，支持更多自定义操作
  - class 的定义和使用

回顾：线性回归

线性回归的定义

- 在统计学中，线性回归 (Linear regression) 是利用称为线性回归方程的最小二乘函数对一个或多个自变量和因变量之间关系进行线性建模的一种回归分析
- 只有一个自变量的情况称为简单回归，大于一个自变量情况的叫做多元回归。

多元线性回归的形式

- 数据样本：$X = [x_1, x_2, ..., x_n]^T$
- 预测目标：Y
- 权值矩阵：$W = [w_1, w_2, ..., w_n]$
- 偏置值：b
- $Y = W * X + b$

回顾：用线性回归求解股票指数预测问题

股票指数预测与线性回归的对应关系

- 利用 300 天前股市的收盘价、最高价与最低价的变化百分比、收盘价与开盘价的变化百分比及成交量预测 300 天后的股市收盘价
- 数据样本 X （每次输入 1 个样本）：
  - 收盘价
  - 最高价与最低价的变化百分比
  - 收盘价与开盘价的变化百分比
  - 成交量
- 预测目标 Y：300 天后股市的收盘价
- 引入可学习的权值矩阵 W 和偏置值 b，有 $Y = W * X + b$

低阶 API 构建线性回归模型

- 程序的结构和标准 Python 实现非常类似
- 使用 tf.GradientTape() 记录并计算梯度
- 使用 tf.keras.optimizers 定义优化器并进行优化

```
a = tf.Variable(initial_value=[[0.], [0.], [0.], [0.]],
    dtype='float64')
b = tf.Variable(initial_value=0., dtype='float64')
variables = [a, b]

num_epoch = 1000
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
for e in range(num_epoch):
    with tf.GradientTape() as tape: # 使用tf.
        GradientTape()记录损失函数的梯度信息
        y_pred = tf.matmul(X_train, a) + b
        loss = tf.reduce_mean(tf.square(y_pred -
            y_train))
    grads = tape.gradient(loss, variables) # 自动计算梯
        度
    optimizer.apply_gradients(grads_and_vars=zip(grads,
```

使用高阶 Keras 接口构建线性回归模型

- Keras 构建和训练深度学习模型的高阶 API
- TF 官网简介三个主要优势：
  - 方便用户使用：
    针对常见用例做出优化的简单而一致的接口
  - 模块化和可组合：
    通过少许限制的可配置模块构建模型
  - 易于扩展：
    可编写自定义构造块，包括新的层、损失函数等。
- 缺点?
  - 相对慢——高层 API
  - 封装后相对不灵活，开发涉及底层原型受限

使用高阶 Keras 接口构建线性回归模型

- tf.keras 是 TF 对 Keras API 规范实现的模型子类 (sub-class)，tf.keras 的底层是 TF
- 使 TF 更易使用，且不失灵活性和性能
- TF 特定功能 (例 Eager Execution、tf.data、Estimator 等) 可支持
- Keras->tf.Keras:import keras->from tensorflow import keras 一般可用
- tf.Keras->Keras: 有 TF 特殊操作就不行

- TF 的 Keras 趋势：
  - TF 2.0 将 Keras 和 Eager Execution 机制作为其简易构建模型的亮点
  - 官方推崇的 slim 也向 keras 靠拢
  - 相比 TF，Keras 接口多 gpu 并行更方便
  - Keras 可将保存 pb 模型文件放入 tensorflow serving 中运行，效率也就只取决于 TF
  - 相比 TF，Keras 上手容易

使用高阶 Keras 接口: 序列模型

- 调用 tf.keras.Sequential() 定义序列模型
- 利用 model.add 追加层

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(units=1, input_dim=4))
    # 可以使用model.summary()看模型的基本情况
```

- 序列模型的问题:
  - 难以共享层及定义分支
  - 难以实现多输入多输出的情况

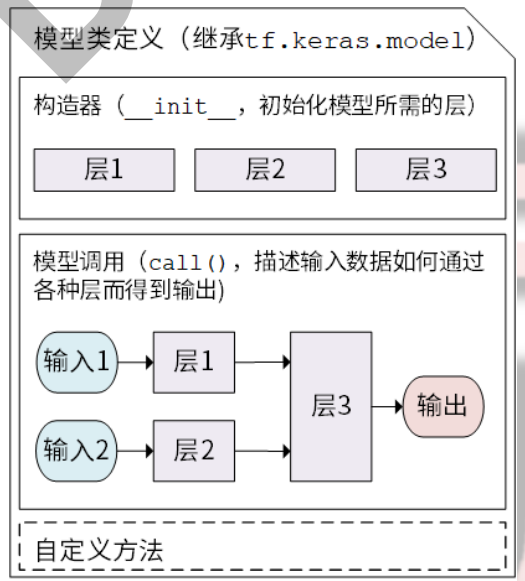
使用高阶 Keras 接口：functional API

- 使用 functional API 可以解决序列模型的一些不足
- 调用 Keras 中的相关接口定义输入层及全连接层
- 利用 Keras.model 将定义的层组织起来

```
inputs = tf.keras.Input(shape=(4,), name='data')
outputs = tf.keras.layers.Dense(units=1, input_dim=4)(
    inputs)
model = tf.keras.Model(inputs=inputs, outputs=outputs,
    name='linear')
```

定义模型类

- 封装更多复杂操作，便于复用
- 需要继承 tf.keras.model 类
- tf.keras.model 类提供了初始化和调用模型的方法，可以继承这些方法方便训练



定义模型类

- 利用模型类实现线性模型
- 没有显式地声明 a 和 b 两个变量及定义线性变换

```
class Linear(tf.keras.Model):
    def __init__(self): # 实例化一个线性层
        super().__init__()
        self.dense = tf.keras.layers.Dense(
                        units=1, input_dim=4, activation=
                            None,
                        kernel_initializer=tf.
                            zeros_initializer(),
                      bias_initializer=tf.
                            zeros_initializer())
    def call(self, input): # 对线性层进行调用并计算
        output = self.dense(input)
        return output
model = Linear()
```

自定义层

- 为什么要自定义层?
  - tf.keras 中提供的构件不能满足一切研究和应用需求
  - 需要实现 tf.keras 中不包括的模型结构时，可以自定义层
  - 自定义层可以与 tf.keras 中的层用相同方式调用

自定义层

- 继承 tf.keras.layers.Layer 类
- 重载 __init__ 、build 和 call 三个方法

```
class LinearLayer(tf.keras.layers.Layer):
        def __init__(self, units):
            super().__init__()
            self.units = units
        def build(self, input_shape):      # 这里
            input_shape 是第一次运行call()时参数inputs的
            形状
            self.w = self.add_variable(name='w',
            shape=[input_shape[-1], self.units],
                initializer=tf.zeros_initializer())
            self.b = self.add_variable(name='b',
            shape=[self.units], initializer=tf.
                zeros_initializer())
        def call(self, inputs):
            y_pred = tf.matmul(inputs, self.w) + self.b
            return y_pred
```

调用自定义层

- 与调用 Keras 层方法类似

```
class LinearModel(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.layer = LinearLayer(units=1)

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

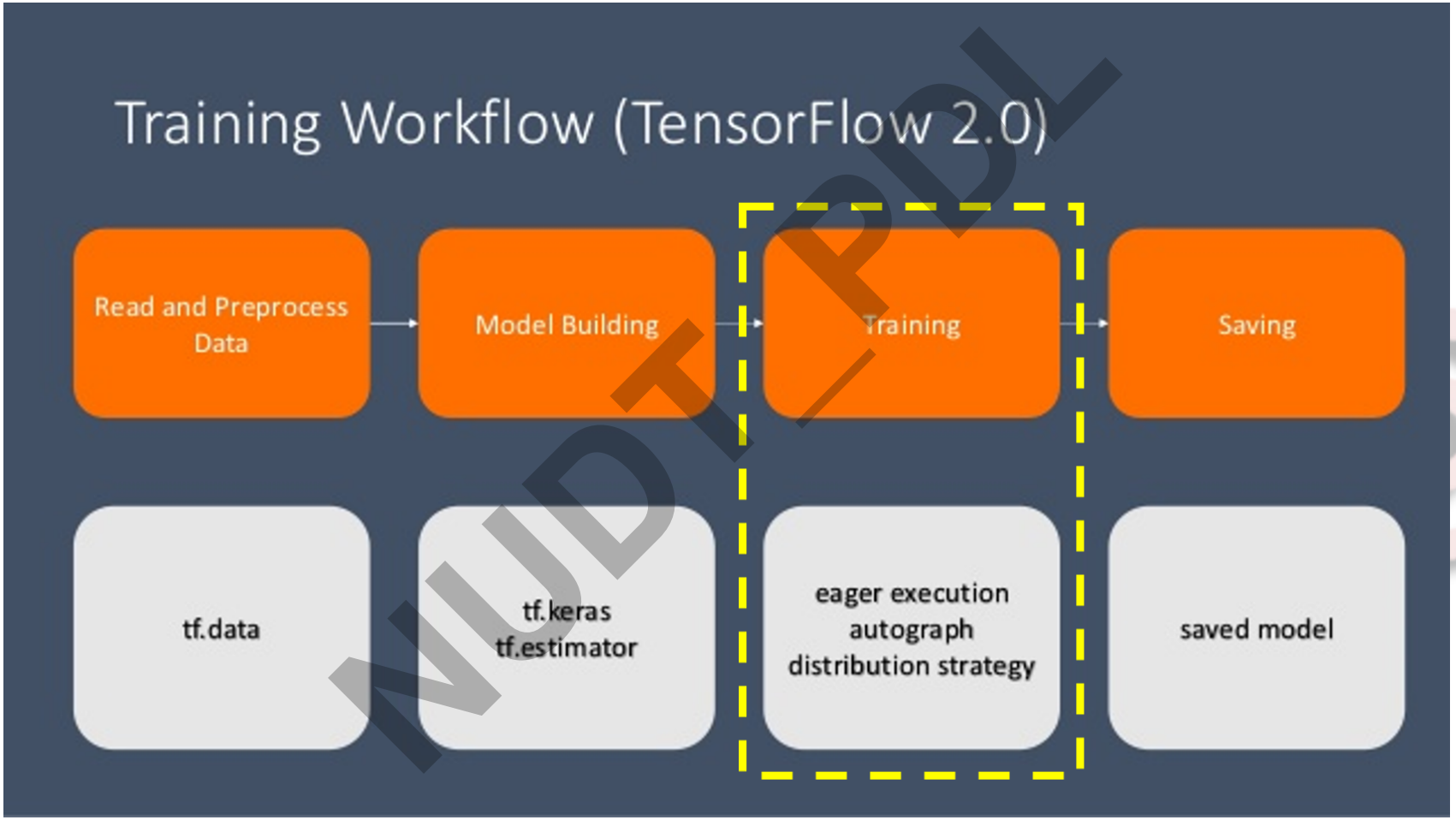
模型类和自定义层

- 模型类和自定义层的缺点：
  - 编程难度大
  - 代码复杂度高
  - 难以调试
- 何时使用模型类和自定义层？
  - 深度学习研究人员需要自定义模型、层和训练方法时
  - 需要复现其他人的研究论文时

模型构建小结

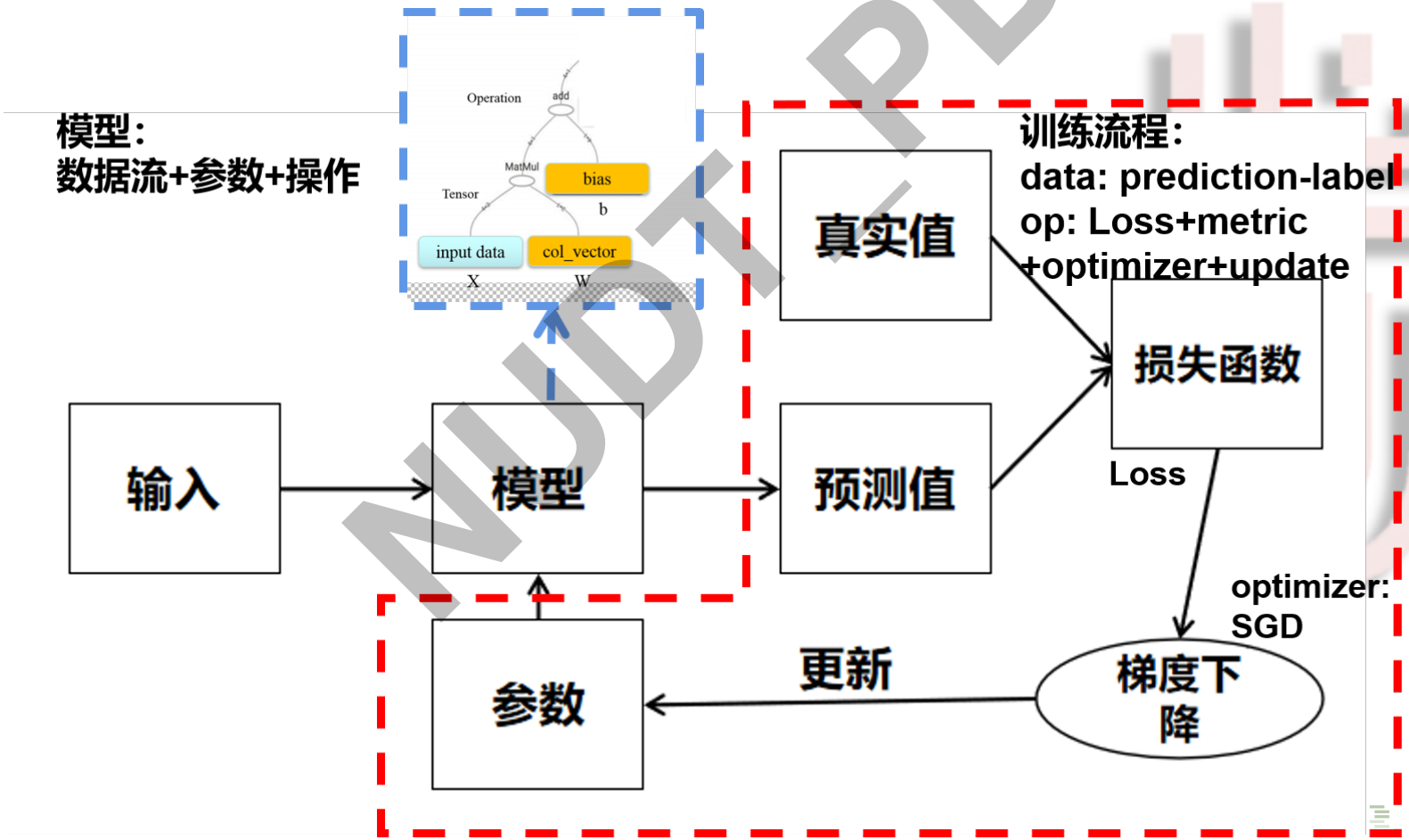
- 低阶 API：仅限于建立及其简单模型及学习训练流程
- 比较简单的模型，实现模块化的搭建，用 sequential
- 构建灵活，复杂一些的模型模块，用 functional api 的方法
- 进行深度学习研究工作，需要对模型的完全控制，用模型类

训练


Training Workflow (TensorFlow 2.0)
Read and Preprocess Data
Model Building
Training
Saving
tf.data
tf.keras
tf.estimator
eager execution
autograph
distribution strategy
saved model

模型训练的基本要素

- 训练数据和标签
- 损失函数和优化器
- 评估指标
- 如何组织训练流程？model.fit 或自定义循环



利用 Keras 中的高级 API 进行训练

- Keras 中包括一系列定义模型训练流程的方法

tf.keras.Model.compile

- 用于配置模型训练，给出有关的参数
  - compile(optimizer='rmsprop', loss=None, metrics=None, loss_weights=None, sample_weight_mode=None, weighted_metrics=None, target_tensors=None, distribute=None, **kwargs)
  - 实践中常用前三个参数：优化器、损失函数、评估指标

tf.keras.Model.fit

- 训练模型。本方法提供了大量与模型训练有关的参数
  - fit(x=None, y=None, batch_size=None, epochs=1, validation_split=0.0, validation_data=None, shuffle=True, ..., **kwargs)
  - 一行代码即可定义大部分训练所需参数

利用 Keras 中的高级 API 进行训练

- 使用 tf.keras.Model.compile 和 tf.keras.Model.fit 实现训练

```
# … … … … … … … … … … … … … … …
model = Linear()

model.compile(
optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
loss='mse', metrics=['mse'])

model.fit(X_train, y_train, epochs=2)
```

自定义循环

- 采用低阶方法定义模型时，不能调用高阶 API 进行训练
- 显式地定义用于模型训练的循环、梯度记录器和损失函数等，显式展示了 model.fit 的内部工作流程
- 可以引入一些更加灵活的、自定义的操作

```
# ……
model = Linear()
optimizer = tf.keras.optimizers.SGD(learning_rate=1e-3)
for e in range(10000):
    with tf.GradientTape() as tape: # 使用tf.
        GradientTape()记录损失函数的梯度信息
        y_pred = model(X_train) # 计算预测值(call)
        loss = 0.5 * tf.reduce_sum(tf.square(y_pred -
            y_train)) # 计算loss
    grads = tape.gradient(loss, variables) # 利用loss计
        算梯度(注意tape)
    optimizer.apply_gradients(grads_and_vars=zip(grads,
        variables)) # 更新参数
```

Gradient Tapes

- tf.GradientTape api：实现自动求导功能
- 只要在 tf.GradientTape() 上下文中执行的操作，都会被记录到“tape”中
- 使用反向自动微分来计算 tape 中记录的操作的梯度。

```
x = tf.ones((2,2))

with tf.GradientTape() as tape:
    tape.watch(x) # x为constant，必须显式监控
    y = tf.reduce_sum(x)
    z = tf.multiply(y,y)

dz_dx = tape.gradient(z, x)
print(dz_dx)
#运行结果如下：
'''
tf.Tensor(
[[8. 8.]
[8. 8.]], shape=(2, 2), dtype=float32)
'''
```

Gradient Tapes

- 默认情况下 GradientTape 的资源会在执行 tf.GradientTape() 后被释放
- 如果想多次计算梯度，需要创建一个持久的 GradientTape。

```
with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)
    y = tf.reduce_sum(x)
    z = tf.multiply(y, y)
dz_dx = tape.gradient(z,x)
print(dz_dx)
dz_dy = tape.gradient(z, y)
print(dz_dy)
#运行结果如下:
'''
tf.Tensor(
[[8. 8.]
[8. 8.]], shape=(2, 2), dtype=float32)
tf.Tensor(8.0, shape=(), dtype=float32)
'''
```

Gradient Tapes

- GradientTape 上下文管理器在计算梯度的同时也会保持梯度
- 可以实现高阶梯度计算

```
x = tf.Variable(1.0) # x为变量，默认被watch，不需要调用
    tape.watch(x)

with tf.GradientTape() as t1:
    with tf.GradientTape() as t2:
        y = x * x * x
    dy_dx = t2.gradient(y, x)
    print(dy_dx)
d2y_d2x = t1.gradient(dy_dx, x)
print(d2y_d2x)
#运行结果如下：
'''
tf.Tensor(3.0, shape=(), dtype=float32)
tf.Tensor(6.0, shape=(), dtype=float32)
'''
```

AutoGraph 机制

- 从 TF 2.0 开始, TF 默认使用动态图模型 (eager execution)
  - 动态图模型：方便灵活，适于科研开发
  - 静态图模型：高效稳定，适于应用部署
  - 能否结合两者优点?
- 利用 AutoGraph 机制在 TensorFlow 2.0 以上的平台上构建传统计算图
  - 给 python 函数加上 @tf.function 装饰器
  - AutoGraph 被自动调用, 将 python 函数转换成可执行的图表示

AutoGraph 机制

- 第一次调用被 @tf.function 装饰的函数时，以下操作将会被执行：
  - 该函数被执行并跟踪。和 Tensorflow 1.x 类似，Eager 会在这个函数中被禁用，因此每个 tf.API 只会定义一个生成 tf.Tensor 输出的节点
  - AutoGraph 用于检测可以转换为等效图表示的 Python 操作 (while→tf.while，for→tf.while，if→tf.cond，assert→tf.assert...)
  - 为了保留执行顺序，在每个语句之后自动添加 tf.control_dependencies，以便在执行第 i+1 行时确保第 i 行已经被执行. 至此计算图已经确定
  - 根据函数名称和输入参数，创建唯一 ID 并将其与定义好的计算图相关联。计算图被缓存到一个映射表中：map[id] = graph
  - 如果 ID 配对上了，之后的函数调用都会直接使用该计算图

在 TensorFlow 2.x 中使用 AutoGraph 机制

- 1. 将计算图代码放入 python 函数

```
def f():
    a = tf.constant([[10,10],[11.,1.]])
    x = tf.constant([[1.,0.],[0.,1.]])
    b = tf.Variable(12.)
    y = tf.matmul(a, x) + b
    return y
print(f().numpy())
#在Eager Execution加持下，此函数可以直接在TF2环境运行。
    运行结果如下:
'''
[[22. 22.]
[23. 13.]]
'''
```

在 TensorFlow 2.x 中使用 AutoGraph 机制

- 2. 追加 @tf.function 修饰符

```
@tf.function
def f():
    a = tf.constant([[10,10],[11.,1.]])
    x = tf.constant([[1.,0.],[0.,1.]])
    b = tf.Variable(12.)  # 在@tf.function函数中不能初
        始化变量
    y = tf.matmul(a, x) + b
    print("PRINT:", y)
    tf.print("TF-PRINT:", y)
    return y
# 当f()被执行的时候，计算图会同时被构建，但是计算不会执
    行
# print命令输出对应Tensor的基本属性，tf.print报错
# 运行结果如下:
'''
PRINT:  Tensor("add:0", shape=(2, 2), dtype=float32)
ValueError: tf.function-decorated function tried to
    create variables on non-first call.
```

在 TensorFlow 2.x 中使用 AutoGraph 机制

- 3. 调用函数：与正常函数方法相同

```
@tf.function
def simple_nn_layer(x, y):
    return tf.nn.relu(tf.matmul(x, y))

x = tf.random.uniform((3, 3))
y = tf.random.uniform((3, 3))
z = simple_nn_layer(x, y)
print(z)
# 运行结果如下:
'''
<tf.Tensor: id=25, shape=(3, 3), dtype=float32, numpy=
array([[0.75023645, 0.19047515, 0.10737072],
[1.1521267 , 0.49491584, 0.19416495],
[0.5541876 , 0.24642248, 0.09543521]], dtype=float32)>
'''
```

AutoGraph 特性

- 从 @tf.function 修饰函数调用的任何函数也将以静态图模式运行

```
def linear_layer(x):
    return 2 * x + 1

@tf.function
def deep_net(x):
    return tf.nn.relu(linear_layer(x))

z = deep_net(tf.constant((1, 2, 3)))
print(z)
# 运行结果如下:
'''
<tf.Tensor: id=39, shape=(3,), dtype=int32, numpy=array
    ([3, 5, 7], dtype=int32)>
'''
```

AutoGraph 特性

- AutoGraph 机制可以将 Python 控制流语句转换为对应的 TensorFlow 版本
- while→tf.while，for→tf.while，if→tf.cond，assert→tf.assert...

```
@tf.function
def sum_even(items):
    s = 0
    for c in items:
        if c % 2 > 0:
            continue
        s += c
    return s

z = sum_even(tf.constant([10, 12, 15, 20]))
print(z)
# 运行结果如下:
'''
<tf.Tensor: id=149, shape=(), dtype=int32, numpy=42>
'''
```

AutoGraph 特性

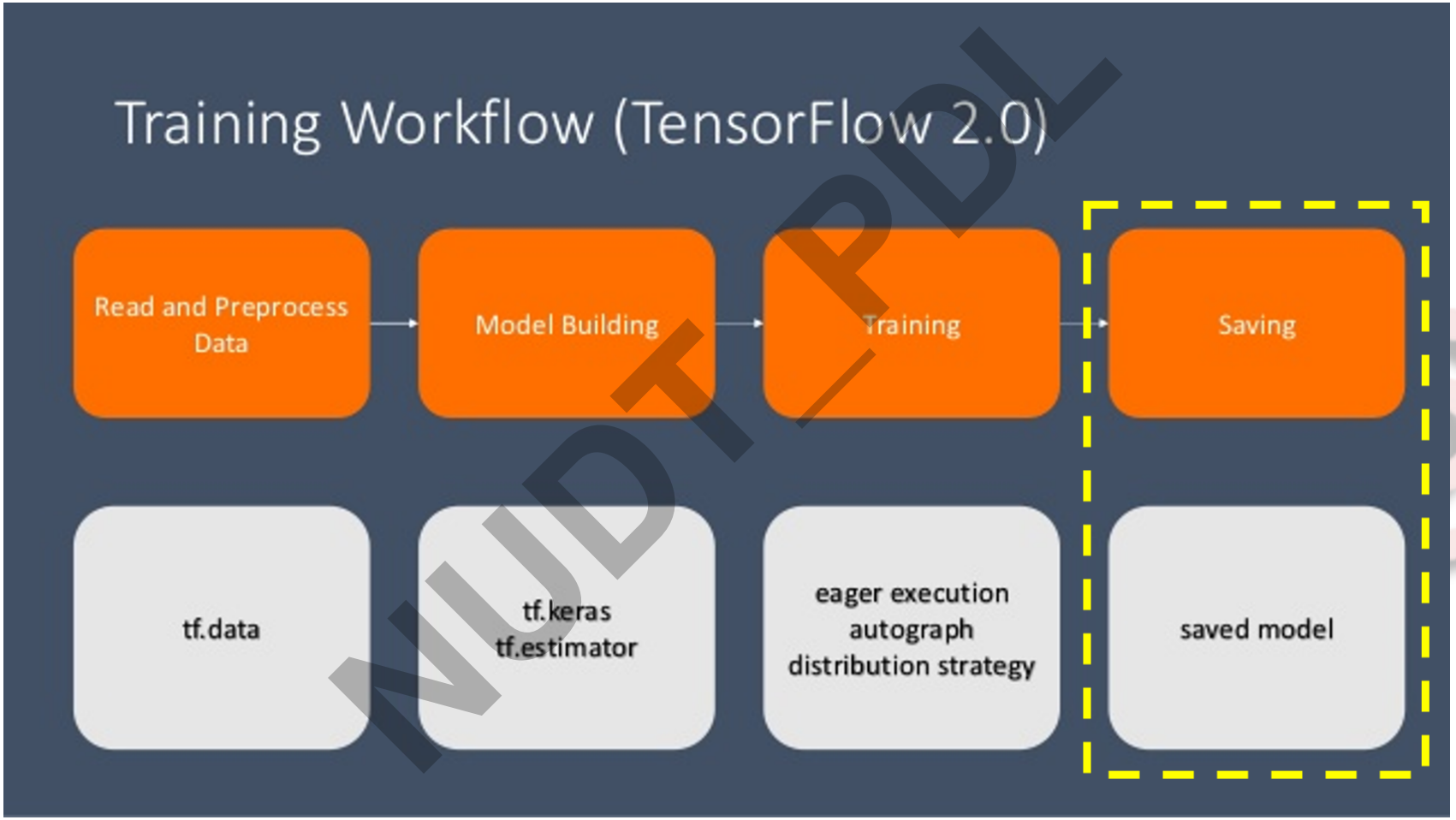
- AutoGraph 机制可以与 Keras 结合
- 可以通过用 @tf.function 修饰自定义 Model 的 call 函数将自定义 Keras 模型转为静态图模型

```
class Linear(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.dense = tf.keras.layers.Dense(
            units=1, input_dim=4, activation=None,
            kernel_initializer=tf.zeros_initializer(),
            bias_initializer=tf.zeros_initializer())
    @tf.function
    def call(self, input):
        output = self.dense(input)
        return output
model = Linear()
model.compile(optimizer=tf.keras.optimizers.SGD(
    learning_rate=0.01),
loss='mse', metrics=['mse'])
model.fit(X_train, y_train, epochs=2)
```

AutoGraph 实例

```
model = Linear()
num_epoch = 1000
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
@tf.function
def train_step(data, labels, model):
    with tf.GradientTape() as tape:
        predictions = model(data)
        loss = tf.reduce_mean(tf.square(predictions -
            labels))#
    gradients = tape.gradient(loss, model.variables)
    optimizer.apply_gradients(grads_and_vars=zip(
        gradients, model.variables))
for e in range(num_epoch):
    train_step(X_train, y_train, model)
```

模型存储



模型存储

- 模型进度可以在训练期间和训练后保存
  - 保存中间结果及最终结果
  - 避免惨案
  - 可以共享模型，方便他人重现工作
- 多种模型保存方法
  - model.fit 中采用回调函数保存的方法
  - 一般保存方法

采用回调函数保存和恢复模型

- 在训练期间和训练结束时自动保存检查点
- tf.keras.callbacks.ModelCheckpoint 用于建立检查点

```
checkpoint_path = "./checkpoints/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)
cp_callback = tf.keras.callbacks.ModelCheckpoint(
    checkpoint_path,
                 save_weights_only=True,
                 verbose=1, period=5) # 每5个迭代保存一
                     次

model = Linear()
model.compile(
     optimizer=tf.keras.optimizers.SGD(learning_rate
         =0.01),
     loss='mse', metrics=['mse'])
model.fit(X_train, y_train, epochs=10, callbacks = [
    cp_callback])
```

国防科技大学 National University of Defense Technology

采用回调函数保存和恢复模型

checkpoint	2020/2/27 22:20	文件	1 KB
cp-0005.ckpt.data-00000-of-00002	2020/2/27 22:20	DATA-00000-OF...	1 KB
cp-0005.ckpt.data-00001-of-00002	2020/2/27 22:20	DATA-00001-OF...	1 KB
cp-0005.ckpt.index	2020/2/27 22:20	INDEX 文件	1 KB
cp-0010.ckpt.data-00000-of-00002	2020/2/27 22:20	DATA-00000-OF...	1 KB
cp-0010.ckpt.data-00001-of-00002	2020/2/27 22:20	DATA-00001-OF...	1 KB
cp-0010.ckpt.index	2020/2/27 22:20	INDEX 文件	1 KB

采用回调函数保存和恢复模型

- 可以在检查点恢复模型
- 仅从权重恢复模型时，必须具有与原始模型具有相同体系结构的模型，由于是相同的模型架构，因此可以共享权重

```
model.fit(X_train, y_train, epochs=10, callbacks = [
    cp_callback])
print(model.variables)

latest = tf.train.latest_checkpoint(checkpoint_dir)
model.load_weights(latest)
print(model.variables)

test_dataset = tf.data.Dataset.from_tensor_slices((
    X_test, y_test))
y_pred_test = model.evaluate(X_test, y_test)
```

采用回调函数保存和恢复模型

```
[<tf.Variable 'linear/dense/kernel:0' shape=(4, 1) dtype=float32, numpy=
array([[315.9945   ],
       [ 14.0329075],
       [  9.922853 ],
       [-11.848215 ]], dtype=float32)>, <tf.Variable 'linear/dense/bias:0'
 shape=(1,) dtype=float32, numpy=array([514.7527], dtype=float32)>]
[<tf.Variable 'linear/dense/kernel:0' shape=(4, 1) dtype=float32, numpy=
array([[315.9945   ],
       [ 14.0329075],
       [  9.922853 ],
       [-11.848215 ]], dtype=float32)>, <tf.Variable 'linear/dense/bias:0'
 shape=(1,) dtype=float32, numpy=array([514.7527], dtype=float32)>]
```

一般保存方法

- model.save_weights：保存模型权重
- model.load_weights：加载模型权重
- model.save：保存整个模型到 hdf5 文件
- model.load_model：加载整个模型，可以在不访问原始 python 代码的情况下使用它

保存和恢复模型权重

```
model.fit(X_train, y_train, epochs=10)
model.save_weights('./checkpoints/my_checkpoint')
print(model.variables)

model.load_weights('./checkpoints/my_checkpoint')
print(model.variables)
```

保存和恢复模型权重

```
[<tf.Variable 'linear/dense/kernel:0' shape=(4, 1) dtype=float32, numpy=
array([[318.72507 ],
       [ 13.153816],
       [  8.862374],
       [ -8.125662]], dtype=float32)>, <tf.Variable 'linear/dense/bias:0' shape=(1,) dtype=float32,
 numpy=array([515.5006], dtype=float32)>]
[<tf.Variable 'linear/dense/kernel:0' shape=(4, 1) dtype=float32, numpy=
array([[318.72507 ],
       [ 13.153816],
       [  8.862374],
       [ -8.125662]], dtype=float32)>, <tf.Variable 'linear/dense/bias:0' shape=(1,) dtype=float32,
 numpy=array([515.5006], dtype=float32)>]
```

保存和恢复整个模型

- Saving the model to HDF5 format requires the model to be a Functional model or a Sequential model. It does not work for subclassed models, because such models are defined via the body of a Python method, which isn't safely serializable.

```
model.fit(X_train, y_train, epochs=10)
model.save('./checkpoints/my_model.h5')
print(model.variables)

new_model = tf.keras.models.load_model('./checkpoints/
    my_model.h5')
print(new_model.variables)
```

保存和恢复整个模型

```
[<tf.Variable 'dense/kernel:0' shape=(4, 1) dtype=float32, numpy=
array([[183.3619   ],
       [  0.9965847],
       [ -5.6818   ],
       [-64.08687  ]], dtype=float32)>, <tf.Variable 'dense/bias:0' shape=(1,) dtype=float32, numpy
=array([460.755], dtype=float32)>]
[<tf.Variable 'dense_1/kernel:0' shape=(4, 1) dtype=float32, numpy=
array([[183.3619   ],
       [  0.9965847],
       [ -5.6818   ],
       [-64.08687  ]], dtype=float32)>, <tf.Variable 'dense_1/bias:0' shape=(1,) dtype=float32, num
py=array([460.755], dtype=float32)>]
```

小结

- 1. Tensorflow 是 Python 接口的深度学习计算库
- 2. TensorFlow 2.x 采用动态图模型
- 3. TF 基本编程模式：数据准备，模型构建，训练，模型保存

使用 TensorFlow

- 使用 Pandas 库导入 csv 数据
- 使用 Numpy 库建立训练集和测试集的 ndarray 矩阵
- 使用 Scikit-learn 库进行数据预处理
- 使用 TensorFlow 库进行线性回归训练与测试
- 使用 matplotlib 进行结果可视化

国防科技大学 National University of Defense Technology

目录

问题：股票指数预测

股票指数预测

- 特征：一定时间段内股市每一天的：
  - 收盘价
  - 最高价与最低价的变化百分比
  - 收盘价与开盘价的变化百分比
  - 成交量
- 预测对象：未来股市的收盘价

问题：股票指数预测

- 原始数据：csv 文件

Date	Open	High	Low	Close	Volume	Ex-Divider	Split Ratio	Adj. Open	Adj. High	Adj. Low	Adj. Close	Adj. Volume
2004/8/19	100.01	104.06	95.96	100.335	44659000	0	1	50.15984	52.19111	48.12857	50.32284	44659000
2004/8/20	101.01	109.08	100.5	108.31	22834300	0	1	50.66139	54.70888	50.4056	54.32269	22834300
2004/8/23	110.76	113.48	109.05	109.4	18256100	0	1	55.55148	56.91569	54.69383	54.86938	18256100
2004/8/24	111.24	111.6	103.57	104.87	15247300	0	1	55.79223	55.97278	51.94535	52.59736	15247300
2004/8/25	104.76	108	103.88	106	9188600	0	1	52.54219	54.16721	52.10083	53.16411	9188600
2004/8/26	104.95	107.95	104.66	107.91	7094800	0	1	52.63749	54.14213	52.49204	54.12207	7094800
2004/8/27	108.1	108.62	105.69	106.15	6211700	0	1	54.21736	54.47817	53.00863	53.23934	6211700
2004/8/30	105.28	105.49	102.01	102.01	5196700	0	1	52.803	52.90832	51.16294	51.16294	5196700

任务：

- 加载原始数据并预处理
- 编写预测程序
- 显示结果

线性回归

线性回归的定义

- 在统计学中，线性回归 (Linear regression) 是利用称为线性回归方程的最小二乘函数对一个或多个自变量和因变量之间关系进行线性建模的一种回归分析
- 只有一个自变量的情况称为简单回归，大于一个自变量情况的叫做多元回归。

多元线性回归的形式

- 数据样本：$X = [x_1, x_2, ..., x_n]^T$
- 预测目标：Y
- 权值矩阵：$W = [w_1, w_2, ..., w_n]$
- 偏置值：b
- $Y = W * X + b$

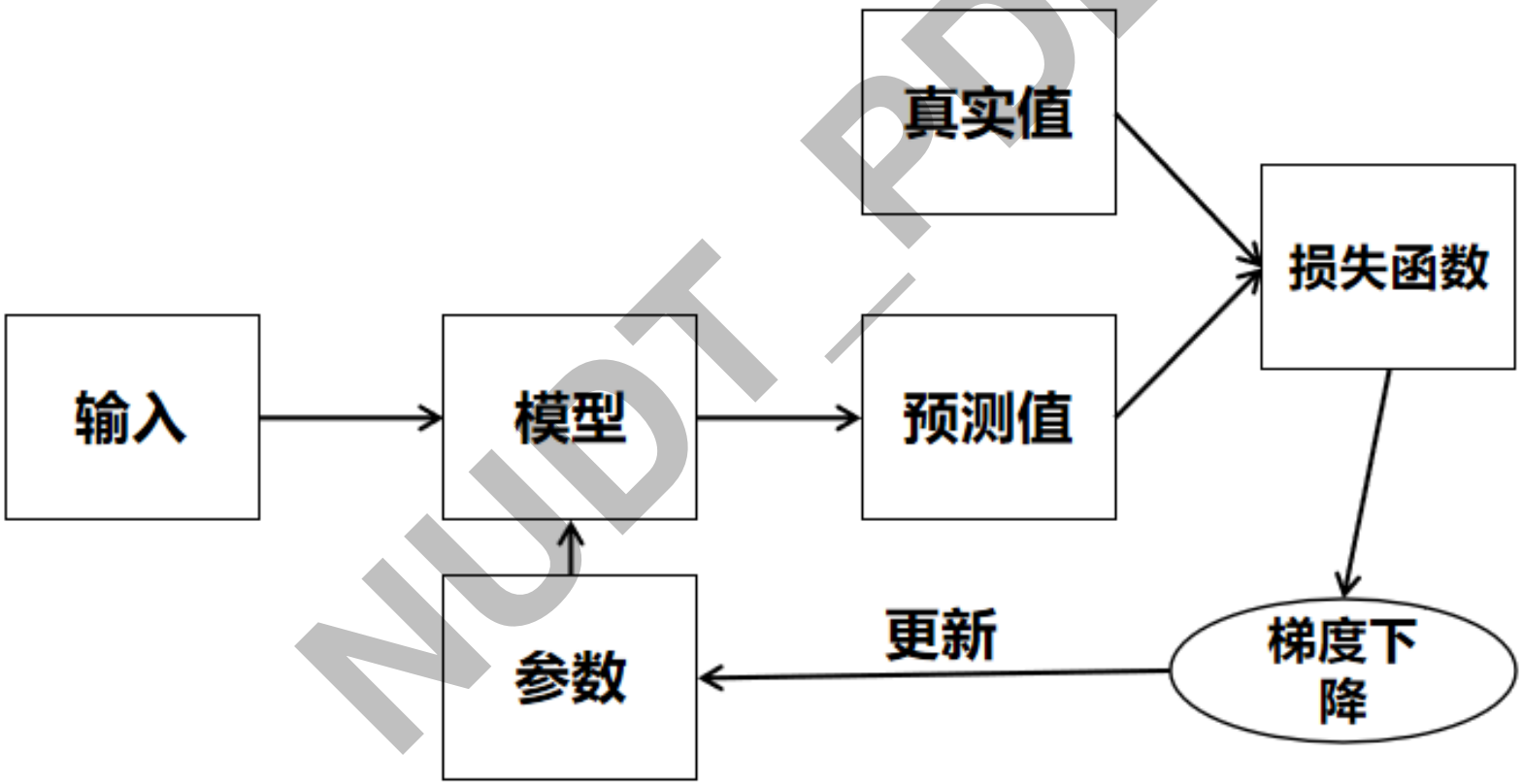
使用 Scikit-learn 处理股票指数预测问题

- 使用 Scikit-learn 进行线性回归:

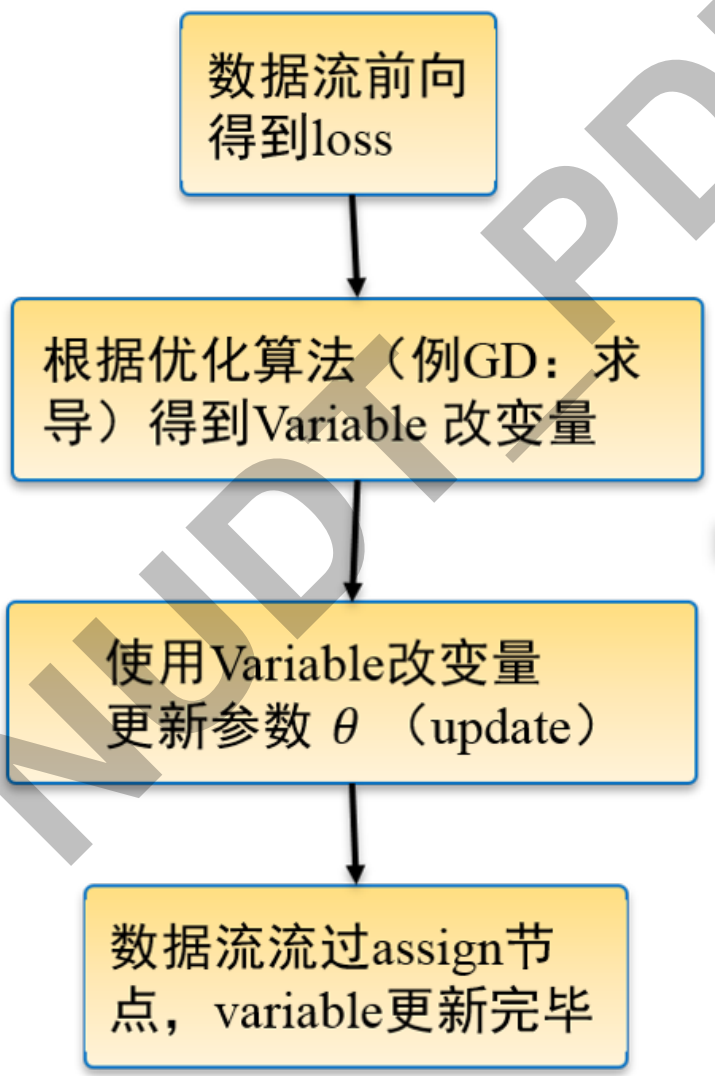
```
X_train, X_test, y_train, y_test=sklearn.
    model_selection.train_test_split(X, y, test_size
    =0.2) #随机划分20%的数据作为测试集

clf = sklearn.linear_model.LinearRegression() # 定义线
    性回归器
clf.fit(X_train, y_train) # 开始训练
accuracy = clf.score(X_test, y_test) # 测试并得到测试集
    性能
```

使用 TensorFlow 处理线性回归问题

- 数据准备，模型构建，训练，模型保存



TensorFlow 优化机制

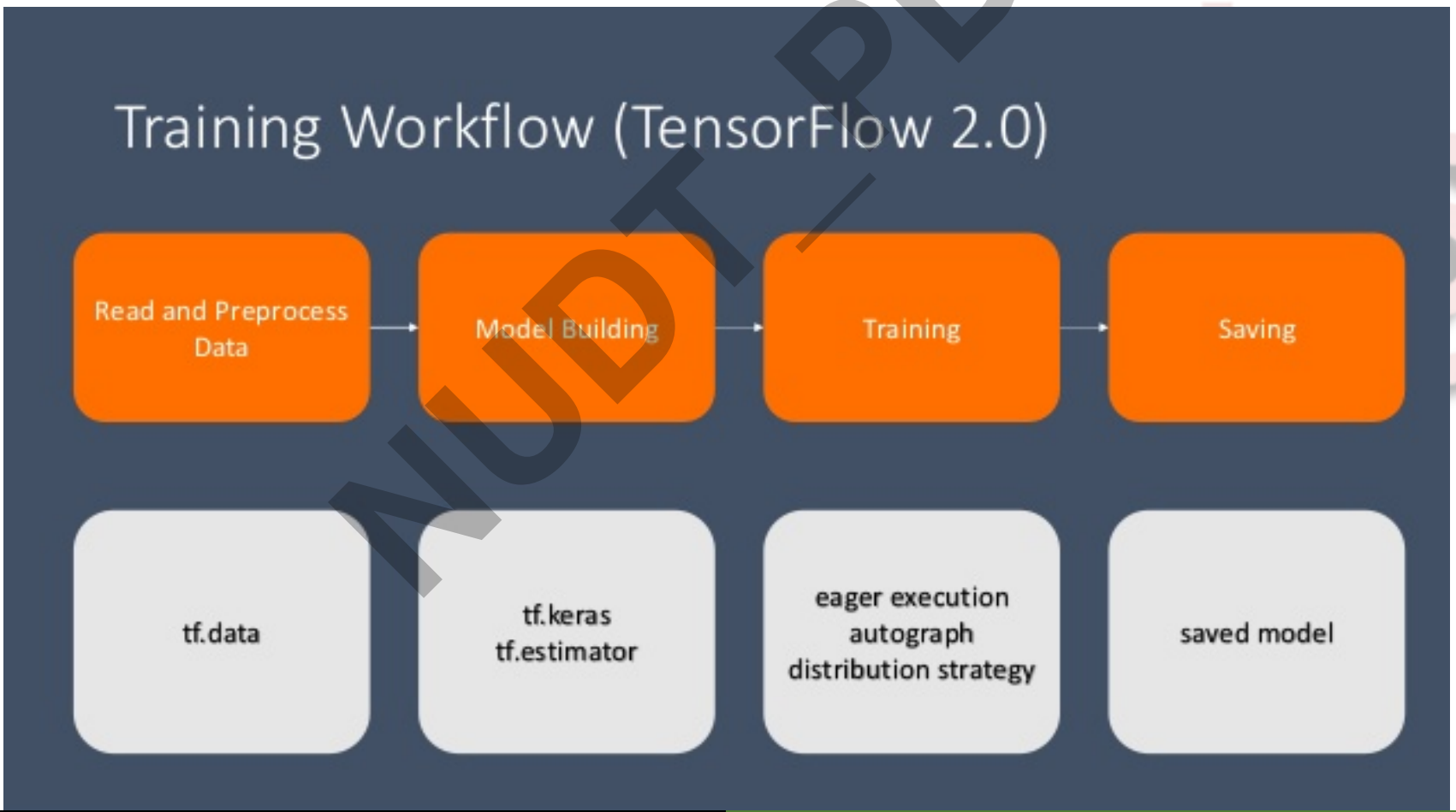


TensorFlow 优化编程模式

- 机器学习优化机制
  - 对象：参数变量
  - 目标：损失函数（最小）
  - 方法：梯度下降等
- TensorFlow 优化编程模式
  - 1. 定义目标函数 (例: 损失函数 loss，模型预测与真值差距)
  - 2. 基于目标函数和优化目标定义优化器
  - 3. 使模型获取数据并调用优化器进行训练

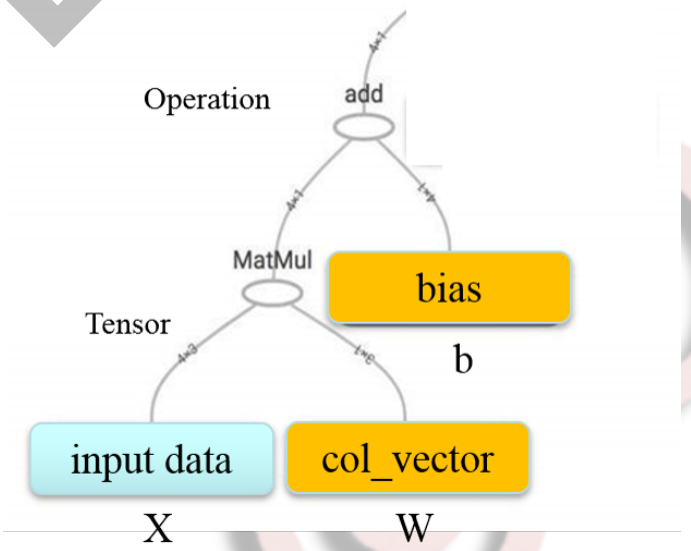
TensorFlow 2.x 基本编程流程

- 数据准备：Numpy, tf.data
- 模型构建：tf.keras
- 训练：eager execution
- 模型保存：savedmodel



1. 股票预测案例数据模型定义

- 输入 X：包含四个维度的特征，即一定时间内每一天股市的：
  - 收盘价
  - 最高价与最低价的变化百分比
  - 收盘价与开盘价的变化百分比
  - 成交量
- 预测对象 Y：未来股市的收盘价
- 权值矩阵 W：1-by-4 矩阵
- 偏置值 b
- 以上信息构成了对计算图的描述 (动态图)



针对股票预测案例定义输入数据

- 直接使用 Numpy 数组作为训练数据：

```
X_train, X_test, y_train, y_test = model_selection.
    train_test_split(X, y, test_size=0.2)
y_train = y_train.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)
# TensorFlow 2.x的模型训练过程可以直接使用numpy数组，因
    此不需要额外操作
```

- 然后需要定义计算模型，并执行训练
  - Y = W * data + b

2、3. 模型构建和训练

TensorFlow 2.x 中可行的几种计算模型定义方式

- 低阶 API 实现
  - 利用 Eager Execution 机制
  - 简单直观，但是对复杂模型力不从心
- 使用高阶 Keras 接口
  - 支持更复杂的网络结构
  - 代码更加简洁优雅
- 定义模型类
  - 支持自定义层
  - class 的定义和使用

使用低阶 API 定义模型并训练

变量定义的基本形式:

- 引用 tensor=tf.Variable(初始化值, 形状, 数据类型, 是否可训练? , 名字, ...)
- w=tf.Variable(initial_value=np.random.randint(10, size=(2,1)), name='col_vector', trainable=True)

为股票预测定义权值、偏置值等变量:

- a = tf.Variable(initial_value=0.)
- b = tf.Variable(initial_value=0.) # 仅指定变量初始化值，不需要显式定义形状等信息，默认可训练

使用低阶 API 定义模型并训练

- 为股票预测定义优化器、线性计算、损失函数和梯度等信息：

```
num_epoch = 10000
optimizer = tf.keras.optimizers.SGD(learning_rate=1e-3)
for e in range(num_epoch):
    # 使用tf.GradientTape()记录损失函数的梯度信息
    with tf.GradientTape() as tape:
        y_pred = a * X_train + b
        loss = 0.5 * tf.reduce_sum(tf.square(y_pred -
            y_train))
    # TensorFlow自动计算损失函数关于自变量（模型参数）
        的梯度
    grads = tape.gradient(loss, variables)
    # TensorFlow自动根据梯度更新参数
    optimizer.apply_gradients(grads_and_vars=zip(grads,
        variables))
```

使用高阶 Keras 接口定义模型并训练

- 无需显式定义变量和梯度等：

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(units=1, input_dim=4))
model.summary()

model.compile(
optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
loss='mse', metrics=['mse'])

model.fit(X_train, y_train, epochs=10)
```

定义模型类

- 利用模型类定义计算模型，更加灵活，且可以实现自定义层和模型：

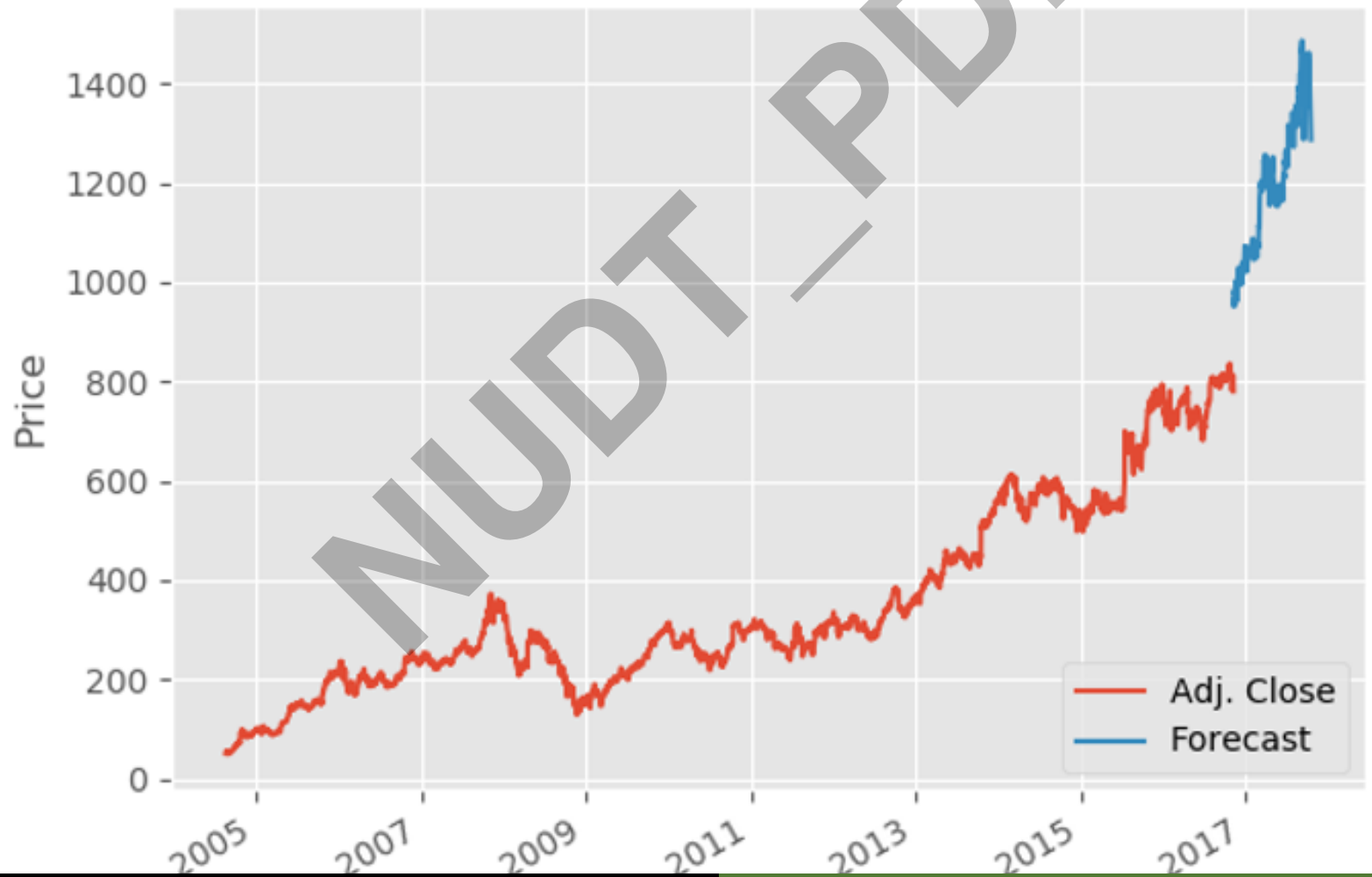
```
class Linear(tf.keras.Model):
    def __init__(self): # 实例化一个线性层
        super().__init__()
        self.dense = tf.keras.layers.Dense(
              units=1, input_dim=4, activation=None,
              kernel_initializer=tf.zeros_initializer()
                  ,
              bias_initializer=tf.zeros_initializer())
    def call(self, input): # 对线性层进行调用并计算
        output = self.dense(input)
        return output
model = Linear()
model.compile(
optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
       loss='mse', metrics=['mse'])
model.fit(X_train, y_train, epochs=10)
```

4. 模型保存

TensorFlow 2.x 中几种模型保存方法

- model.fit 中采用回调函数保存的方法
- 一般保存方法：保存权重或保存整个模型

可视化结果



国防科技大学 National University of Defense Technology

目录

国防科技大学 National University of Defense Technology

总结

- TensorFlow 基本概念
- TensorFlow 编程流程
- 用 TensorFlow 实现基于线性回归的股价预测

国防科技大学 National University of Defense Technology

下集预告

- 神经网络原理
- 多层神经网络
- TensorFlow 实现神经网络