《使用 Python、AI 与 Kubernetes 构建可扩展图像分类服务》实验手册

一、实验目标

1. 掌握使用 Python 和 TensorFlow 构建简单图像分类 AI 模型的方法。
2. 熟练运用 PyCharm 进行 Python 代码的开发与调试。
3. 理解 Kubernetes 的核心概念，并能够使用 Kubernetes 对 AI 服务进行容器化部署与管理，实现服务的可扩展性。

二、实验环境

4. **操作系统**：Windows、Linux 或 macOS 均可。
5. **软件工具**：
    - PyCharm：用于 Python 代码开发。
    - Python 3.x：项目开发的基础环境。
    - TensorFlow、Keras、OpenCV、Flask、Docker、Minikube、kubectl：相关依赖包与工具。

三、实验步骤

第一节课实验：Python 与 AI 模型构建

（一）课程导入

6. **图像分类应用场景**
    - 医疗影像诊断：图像分类助力医生从 X 光、CT、MRI 等影像里找出病变区域，比如在肺部 CT 影像中判断有无肿瘤，为疾病诊断提供重要参考。
    - 自动驾驶领域：借助图像分类技术，车辆可识别前方行人、车辆、交通标志等物体，从而做出驾驶决策，保障行车安全。
7. **Kubernetes 的重要性**
    - **高可用性**：Kubernetes 通过多副本部署及自动故障检测与恢复机制，确保应用始终可用。例如，若某个运行应用的容器出现故障，Kubernetes 会自动启动新容器替代，维持服务不中断。

- **弹性扩展**：依据实际业务负载，Kubernetes 能自动调整容器数量。流量高峰时增加容器以提升处理能力，流量低谷时减少容器以节省资源，提高资源利用率。

（二）环境搭建

8. **安装软件和依赖包**
   - **PyCharm**：访问 JetBrains 官网，下载适配操作系统的 PyCharm 版本并完成安装。
   - **Python 3.x**：从 Python 官方网站下载对应操作系统的安装包进行安装，安装过程勾选将 Python 添加到系统路径，方便在命令行使用 Python 命令。
   - **相关依赖包**：
     - **TensorFlow**：在命令行执行 pip install tensorflow 安装，这是广泛使用的深度学习框架，用于构建、训练和部署深度学习模型。
     - **Keras**：通常随 TensorFlow 安装一同完成，它是基于 TensorFlow 的高级神经网络 API，简化深度学习模型构建流程。
     - **OpenCV**：在命令行执行 pip install opencv - python 安装，虽本项目处理 MNIST 数据集无需此库，但处理实际图像数据时非常有用，用于计算机视觉任务，如图像读取、处理和显示等。
9. **创建 PyCharm 项目**
   - 打开 PyCharm，选择“Create New Project”。
   - 在弹出窗口中，设定项目存储路径，命名项目（如“ImageClassificationProject”），并选择已安装的 Python 解释器，点击“Create”完成项目创建。

（三）图像分类 AI 模型开发

10. **深度学习图像分类原理**
    - **卷积神经网络（CNN）**：专为处理网格结构数据（如图像）设计的神经网络，主要由卷积层、池化层和全连接层构成。
      - **卷积层**：利用卷积核在图像上滑动，对局部区域进行特征提取，卷积核权重通过训练学习得到，不同卷积核可提取如边缘、纹理等不同特征。
      - **池化层**：一般用于减少数据维度并保留关键特征，常见方式有最大池化（选取池化窗口内最大值输出）和平均池化（计算窗口内平均值输出），可降低计算量，防止过拟合。
      - **全连接层**：在卷积层和池化层完成特征提取与降维后，将数据展开为一维向量，通过全连接层综合分析这些特征，输出最终分类结果。
11. **代码实现讲解与操作**

- **数据加载**:

在 PyCharm 项目中新建 Python 文件（如 model_dev.py），输入以下代码:

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

执行该代码，mnist.load_data()会从 Keras 内置 MNIST 数据集中加载 60,000 张手写数字 0 - 9 的训练图像及对应标签，以及 10,000 张测试图像及对应标签，每张图像大小为 28x28 像素。

- **数据预处理**:

继续在 model_dev.py 中添加代码:

```
train_images = train_images.reshape((-1, 28, 28, 1)).astype('float32') / 255.0
test_images = test_images.reshape((-1, 28, 28, 1)).astype('float32') / 255.0
```

reshape 函数将二维图像数据转为适合 CNN 输入的四维张量，-1 使系统自动计算该维度大小，(28, 28, 1)分别表示图像高度、宽度和通道数（MNIST 为灰度图，通道数 1）。除以 255.0 将像素值归一化到 0 - 1 范围，利于模型训练收敛。

- **模型构建**:

在 model_dev.py 中继续添加代码构建模型:

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

Sequential 模型是 Keras 中简单的线性堆叠模型。第一个 Conv2D 层含 32 个(3, 3)大小卷积核，激活函数 relu，输入形状(28, 28, 1)，负责提取低级特征。MaxPooling2D 层对卷积层输出进行最大池化，窗口(2, 2)，降低数据维度。第二个 Conv2D 层和 MaxPooling2D 层进一步提取和压缩特征。Flatten 层将多维数据展平为一维向量，方便输入全连接层。第一个 Dense 层 64 个神经元，激活函数 relu，处理展平后的特征。最后一个 Dense 层 10 个神经元，对应 10 个数字类别，激活函数 softmax，输出每个类别的概率分布。

- **模型编译**:

在 model_dev.py 中添加编译代码:

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

optimizer='adam'选用 Adam 优化器，它是常用的自适应学习率优化算法，助力模型更快收敛。loss='sparse_categorical_crossentropy'选择稀疏分类交叉熵损失函数，适用于多分类且标签为整数的情况（MNIST 标签为 0 - 9 整数）。metrics=['accuracy']指定训练和评估时监控指标为准确率。

- **模型训练**:

在 model_dev.py 中添加训练代码:

```
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

model.fit 用于训练模型，train_images 和 train_labels 为训练数据与标签。epochs=5 表示模型对整个训练数据集遍历 5 次。batch_size=64 指每次训练从训练数据集中取 64 个样本进行梯度计算与参数更新。小批量有助于更好收敛但增加训练时间，大批量可加快训练但可能影响收敛效果。

- **模型评估**:

在 model_dev.py 中添加评估代码:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test accuracy: {test_acc}")
```

model.evaluate 使用测试数据集 test_images 和 test_labels 评估训练好的模型，返回测试集上的损失值 test_loss 和准确率 test_acc，最后打印测试准确率评估模型性能。

（四）模型保存

在 model_dev.py 中添加保存模型代码:

```
model.save('mnist_model.h5')
```

model.save 将训练好的模型保存为 HDF5 格式文件 mnist_model.h5，该文件包含模型结构、权重及训练配置等信息，方便后续加载使用。

第二节课实验：Kubernetes 部署与管理

（一）Kubernetes 基础概念讲解

12. **Pod**：Kubernetes 中最小可部署和管理的计算单元，一个 Pod 可含一个或多个紧密相关容器，这些容器共享网络命名空间、存储卷等资源。如 Web 应用中，一个 Pod 可能包含 Web 服务器容器和数据库客户端容器，协同提供完整服务。Pod 内容器通常一同启动、停止，在同一宿主机上运行，通过 localhost 通信。
13. **Service**：为一组功能相同的 Pod 提供固定网络入口，可将外部流量均匀分发到后端 Pod，实现负载均衡。例如，多个 Pod 运行图像分类服务时，Service 能将用户请求转发到其中一个 Pod 处理，提升服务可用性与处理能力。Service 有 ClusterIP（默认，用于集群内部通信）、NodePort（在各节点开放端口，允许外部通过节点 IP 和端口访问）、LoadBalancer（借助云提供商负载均衡器暴露服务到外部）等类型。
14. **Deployment**：用于管理 Pod 生命周期，包括创建、更新、扩展和收缩 Pod。通过定义 Deployment，可指定 Pod 副本数量、使用的容器镜像等。比如，修改副本数量可轻松实现服务水平扩展或收缩，以适应不同业务负载。更新容器镜像时，Deployment 会自动执行滚动升级，逐步替换旧版本 Pod 为新版本，确保服务升级不中断。
15. **容器化概念**：将应用程序及其依赖打包成独立、可移植单元（容器）的过程。容器包含运行应用所需的代码、运行时环境、系统工具和库等组件。在 Kubernetes 中，容器作为部署和管理应用的基本单元，具有轻量级、可移植、隔离性好等优点，不同容器可在同一宿主机上运行且相互隔离，使同一基础设施能高效运行多个不同应用。

（二）容器化 AI 服务

16. **Dockerfile 编写**

在项目根目录创建 Dockerfile 文件，输入以下内容：

```
FROM python:3.8
WORKDIR /app
COPY requirements.txt.
RUN pip install -r requirements.txt
COPY. /app
CMD ["python", "app.py"]
```

FROM python:3.8 指定基础镜像为官方 Python 3.8 镜像，含 Python 3.8 运行环境和基本系统工具。WORKDIR /app 设置容器内工作目录为/app。COPY requirements.txt.将本地项目的 requirements.txt 文件复制到容器工作目录。RUN pip install -r requirements.txt 在容器内按

requirements.txt 依赖列表安装项目所需 Python 包（tensorflow 和 numpy）。COPY. /app 将本地项目所有文件复制到容器工作目录。CMD ["python", "app.py"]指定容器启动时运行 app.py 文件。

2. **requirements.txt 文件**

在项目根目录创建 requirements.txt 文件，输入：

```
tensorflow
numpy
```

该文件列出项目运行所需 Python 依赖包，未指定版本则安装最新版。本项目中，tensorflow 用于加载和运行训练好的模型，numpy 用于数值计算。

3. **app.py 文件编写**

在项目根目录创建 app.py 文件，输入以下代码：

```
import tensorflow as tf
import numpy as np
from flask import Flask, request, jsonify
app = Flask(__name__)
model = tf.keras.models.load_model('mnist_model.h5')
@app.route('/predict', method=['POST'])
def predict():
    data = request.get_json(force=True)
    image = np.array(data['image']).reshape((1, 28, 28, 1)).astype('float32') / 255.0
    prediction = model.predict(image).tolist()
    return jsonify({'prediction': prediction})
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

导入必要库，tensorflow 加载和运行模型，numpy 处理数值数据，Flask 创建 HTTP 服务接口。创建 Flask 应用实例 app，用 tf.keras.models.load_model 加载 mnist_model.h5 模型。定义/predict 路由处理 POST 请求，接收请求后用 request.get_json(force=True)获取 JSON 数据，将 image 字段转为 NumPy 数组并进行与训练时相同的预处理，用加载模型预测并将结果转为列表，通过 jsonify 以 JSON 格式返回。最后，app.run(host='0.0.0.0', port=5000)启动 Flask 应用，监听所有网络接口的 5000 端口。

4. **构建容器镜像**

在项目根目录打开命令行终端，执行命令 docker build -t mnist - classifier:v1.。docker build 是 Docker 构建镜像命令，-t 指定镜像标签，mnist - classifier 为仓库名，v1 为版本号标签，. 表示构建上下文为当前目录，Docker 会在此目录查找 Dockerfile 并按其指令构建镜像。

（三）Kubernetes 部署

17. **安装和配置 Kubernetes 环境（以 Minikube 为例）**
    - **安装 Minikube**：
      - **Linux 系统**：在命令行执行以下命令下载并安装：

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

- **其他系统**：根据 Minikube 官方网站针对对应操作系统的指引完成安装。
- **启动 Minikube**：在命令行执行 minikube start，Minikube 会创建单节点 Kubernetes 集群，启动过程可能需下载镜像文件，依网络情况等待。
- **配置 kubectl**：kubectl 是 Kubernetes 命令行工具，Minikube 启动后自动配置其与本地 Minikube 集群通信。可执行 kubectl config current - context 查看当前上下文，确保为 Minikube 集群。

18. **创建 Deployment 配置文件（mnist - deployment.yaml）**

在项目根目录创建 mnist - deployment.yaml 文件，输入以下内容：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mnist - classifier - deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mnist - classifier
  template:
```

```
  metadata:
    labels:
      app: mnist - classifier
  spec:
    containers:
    - name: mnist - classifier
      image: mnist - classifier:v1
      ports:
      - containerPort: 5000
```

apiVersion: apps/v1 指定 Kubernetes API 版本，适用于大多数 Kubernetes 版本的 Deployment 资源。kind: Deployment 表明资源类型为 Deployment。metadata.name 为 Deployment 命名为 mnist - classifier - deployment。spec.replicas 指定创建 3 个 Pod 副本，提高服务可用性与处理能力。spec.selector.matchLabels 通过 app: mnist - classifier 标签选择相关 Pod。spec.template 定义 Pod 模板，metadata.labels 为 Pod 添加与选择器一致的标签方便管理，spec.containers 定义 Pod 内运行的容器，containers.name 指定容器名，containers.image 指定使用之前构建的 mnist - classifier:v1 镜像，containers.ports.containerPort 指定容器内部监听端口 5000，与 app.py 中 Flask 应用监听端口一致。

3. **创建 Service 配置文件（mnist - service.yaml）**

在项目根目录创建 mnist - service.yaml 文件，继续补充完整内容：

```
apiVersion: v1
kind: Service
metadata:
  name: mnist - classifier - service
spec:
  selector:
    app: mnist - classifier
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
```

```
type: LoadBalancer
```

apiVersion: v1 指定 Kubernetes API 版本适用于 Service 资源。kind: Service 表明资源类型为 Service。metadata.name 为 Service 命名为 mnist - classifier - service。spec.selector 通过 app: mnist - classifier 标签选择对应的 Pod。spec.ports 定义服务端口，protocol: TCP 指定协议为 TCP，port: 80 是服务对外暴露的端口，targetPort: 5000 是 Pod 内部容器监听的端口。type: LoadBalancer 指定 Service 类型为 LoadBalancer，使用云提供商的负载均衡器将服务暴露到外部（在 Minikube 环境中，可通过 minikube service 命令模拟外部访问）。

4. **部署应用并验证**

- 在项目根目录命令行终端执行 kubectl apply -f mnist - deployment.yaml 和 kubectl apply -f mnist - service.yaml，分别创建 Deployment 和 Service 资源。
- 执行 kubectl get pods 查看 Pod 状态，确保 3 个 Pod 都处于运行（Running）状态。执行 kubectl get services 查看 Service 信息，获取服务的访问地址。
- 使用工具（如 Postman）发送 POST 请求到服务地址的/predict 接口，