# 《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

1. **操作系统**：Windows、Linux或macOS均可。
2. **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

1. **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。
2. **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

1. **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。
2. **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

1. **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。
2. **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

1. **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。
2. **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。
3. **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。
4. **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

1. **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

1. **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。

1. **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，