HOME ASSIGNMENT 2

NAME: JASWANTH REDDY DONAPATI

ID: 700757646

**1-ANS)**

**(a) Elasticity and Scalability in the Context of Cloud Computing for Deep Learning:**

* **Elasticity:**  
  Elasticity in cloud computing refers to the ability of a system to dynamically adjust its resources (such as processing power, storage, and memory) based on the current demand. In the context of deep learning, elasticity ensures that as computational needs increase (e.g., during model training), the cloud platform can automatically allocate more resources. Similarly, when the demand decreases (e.g., during inference or testing), the system can scale down the resources, ensuring cost-efficiency. This dynamic allocation enables deep learning models to handle variable workloads without manual intervention.
* **Scalability:**  
  Scalability refers to the capability of a system to handle an increasing amount of workload or its potential to be enlarged to accommodate that growth. In deep learning, scalability allows users to expand their compute resources (such as adding more GPUs or CPUs) as the size of the dataset, model complexity, or training duration increases. For example, scaling up enables faster model training, while scaling out (i.e., distributing tasks across multiple machines) can improve parallel processing for large-scale data analysis, thus enhancing the deep learning.

**(b) Comparison of AWS SageMaker, Google Vertex AI, and Microsoft Azure Machine Learning Studio for Deep Learning:**

| **Feature** | **AWS SageMaker** | **Google Vertex AI** | **Microsoft Azure Machine Learning Studio** |
| --- | --- | --- | --- |
| **Deep Learning Frameworks** | SageMaker supports TensorFlow, PyTorch, MXNet, and more. | Vertex AI supports TensorFlow, PyTorch, and JAX. | Azure ML supports TensorFlow, PyTorch, Scikit-learn, and more. |
| **Model Training** | Provides managed training with automatic model tuning, multi-instance training, and distributed training. | Offers custom training, AutoML for model tuning, and scalable distributed training. | Offers distributed training using popular frameworks and automated hyperparameter tuning. |
| **Ease of Use** | Fully managed environment with easy setup for users of all skill levels, from beginner to expert. | Simplified user interface with built-in AutoML for easy model creation and deployment. | User-friendly interface with drag-and-drop features, good for non-technical users. |
| **Integration with Other Services** | Seamless integration with other AWS services (e.g., S3 for data storage, Lambda for serverless computing). | Strong integration with other Google Cloud services like BigQuery and Dataflow for big data processing. | Deep integration with Microsoft services like Power BI, Azure Databricks, and Azure Synapse Analytics. |
| **AutoML and Hyperparameter Tuning** | SageMaker provides automatic model tuning (Hyperparameter Optimization) and AutoML features. | Vertex AI’s AutoML provides high-level model creation, with customizations available. | Azure ML offers AutoML and Hyperparameter tuning features for easy model optimization. |
| **Compute and Hardware Options** | Offers diverse compute options like CPU, GPU, and distributed multi-GPU configurations. | Provides high-performance GPUs, TPUs, and cloud-based clusters for scaling. | Offers a range of compute options including GPU, FPGA, and multi-node clusters. |
| **Deployment and Inference** | Allows easy deployment with built-in endpoints for real-time inference, batch predictions, and hosting models. | Simplifies deployment with Vertex AI endpoints, enabling model serving, batch predictions, and version control. | Offers various deployment options for web services, IoT devices, and containerized environments. |
| **Cost Structure** | Pay-as-you-go pricing, with free tier options for beginners. | Flexible pay-as-you-go pricing based on resource usage and AI model complexity. | Offers a pay-as-you-go pricing model with options for reserved instances. |
| **Security and Compliance** | Integrated with AWS’s robust security features like VPC, IAM roles, and encryption at rest. | Strong security and compliance features, with built-in encryption and access control. | Provides enterprise-grade security with features like role-based access control, encryption, and compliance standards. |

**Conclusion:**

* **AWS SageMaker** is best suited for users already integrated into the AWS ecosystem, offering robust deep learning frameworks and scalable computing resources, ideal for large-scale projects.
* **Google Vertex AI** shines for users who need advanced AI capabilities like Tensor Processing Units (TPUs) and integration with Google’s big data tools and is perfect for machine learning tasks that require seamless scalability.
* **Microsoft Azure Machine Learning Studio** is a great option for businesses looking for an easy-to-use platform with drag-and-drop features, good integration with Microsoft tools, and flexible model deployment options, making it ideal for enterprise solutions.

**2-ANS)**

CODE:

import numpy as np

import tensorflow as tf

# Step 1: Define the 5x5 input matrix

# The input matrix is a simple 5x5 matrix with values from 1 to 25.

input\_matrix = np.array([[1, 2, 3, 4, 5],

[6, 7, 8, 9, 10],

[11, 12, 13, 14, 15],

[16, 17, 18, 19, 20],

[21, 22, 23, 24, 25]])

# Step 2: Define the 3x3 kernel

# The kernel is a 3x3 matrix. This kernel is designed to perform edge detection.

kernel = np.array([[1, 0, -1],

[1, 0, -1],

[1, 0, -1]])

# Reshaping input matrix and kernel for TensorFlow/Keras compatibility

# For TensorFlow's `conv2d` function, the input tensor needs to have shape (batch\_size, height, width, channels).

input\_tensor = np.expand\_dims(input\_matrix, axis=0) # Shape: (1, 5, 5) -- Add batch dimension

input\_tensor = np.expand\_dims(input\_tensor, axis=-1) # Shape: (1, 5, 5, 1) -- Add channels dimension

# Reshaping the kernel for TensorFlow compatibility

kernel\_tensor = np.expand\_dims(kernel, axis=-1) # Shape: (3, 3, 1) -- Add input channels dimension

kernel\_tensor = np.expand\_dims(kernel\_tensor, axis=-1) # Shape: (3, 3, 1, 1) -- Add output channels dimension

# Convert the tensors to float32

input\_tensor = tf.cast(input\_tensor, tf.float32) # Cast input tensor to float32

kernel\_tensor = tf.cast(kernel\_tensor, tf.float32) # Cast kernel tensor to float32

# Step 3: Perform convolution with different configurations

# Stride = 1, Padding = 'VALID'

# 'VALID' padding means no padding, and the output will be smaller than the input.

conv\_valid\_stride\_1 = tf.nn.conv2d(input\_tensor, kernel\_tensor, strides=[1, 1, 1, 1], padding='VALID')

# Stride = 1, Padding = 'SAME'

# 'SAME' padding adds zero padding so that the output dimensions are the same as the input when stride is 1.

conv\_same\_stride\_1 = tf.nn.conv2d(input\_tensor, kernel\_tensor, strides=[1, 1, 1, 1], padding='SAME')

# Stride = 2, Padding = 'VALID'

# Stride of 2 skips every other pixel, and 'VALID' padding reduces the output size.

conv\_valid\_stride\_2 = tf.nn.conv2d(input\_tensor, kernel\_tensor, strides=[1, 2, 2, 1], padding='VALID')

# Stride = 2, Padding = 'SAME'

# With stride 2 and 'SAME' padding, TensorFlow adds padding to maintain an output size compatible with the input size.

conv\_same\_stride\_2 = tf.nn.conv2d(input\_tensor, kernel\_tensor, strides=[1, 2, 2, 1], padding='SAME')

# Step 4: Print the output feature maps for each case

# Print output for Stride = 1, Padding = 'VALID'

print("Output Feature Map (Stride = 1, Padding = 'VALID'):")

print(conv\_valid\_stride\_1.numpy().squeeze()) # .numpy() converts the result to a NumPy array

# Print output for Stride = 1, Padding = 'SAME'

print("\nOutput Feature Map (Stride = 1, Padding = 'SAME'):")

print(conv\_same\_stride\_1.numpy().squeeze()) # .squeeze() removes single-dimensional entries from the shape

# Print output for Stride = 2, Padding = 'VALID'

print("\nOutput Feature Map (Stride = 2, Padding = 'VALID'):")

print(conv\_valid\_stride\_2.numpy().squeeze())

# Print output for Stride = 2, Padding = 'SAME'

print("\nOutput Feature Map (Stride = 2, Padding = 'SAME'):")

print(conv\_same\_stride\_2.numpy().squeeze())

**OUTPUT:**

A white text with black text

AI-generated content may be incorrect.

**3 ANS)**

**Task 1: Implement Edge Detection Using Sobel Filter**

CODE:

import cv2

import numpy as np

import matplotlib.pyplot as plt

import os

import urllib.request

# Check if the image exists, if not download it

image\_path = 'sample\_image.jpg'

if not os.path.exists(image\_path):

print(f"'{image\_path}' not found. Downloading...")

url = "https://upload.wikimedia.org/wikipedia/commons/thumb/b/b6/SIPI\_Jelly\_Beans\_4.1.07.tiff/lossy-page1-256px-SIPI\_Jelly\_Beans\_4.1.07.tiff.jpg"

try:

urllib.request.urlretrieve(url, image\_path)

print(f"'{image\_path}' downloaded successfully.")

except Exception as e:

print(f"Error downloading '{image\_path}': {e}")

exit()

else:

print(f"'{image\_path}' found.")

# Step 1: Load a grayscale image

# Load the image using OpenCV's imread function. 'sample\_image.jpg' should be replaced with the path to your image.

# The image is loaded in grayscale mode (cv2.IMREAD\_GRAYSCALE).

image = cv2.imread(image\_path, cv2.IMREAD\_GRAYSCALE)

# Step 2: Apply Sobel filter for edge detection in the x-direction (Sobel-X)

# Sobel filter is applied to the image to detect horizontal edges. dx=1 and dy=0 indicates Sobel-X.

# ksize=3 specifies the size of the kernel (3x3 in this case).

sobel\_x = cv2.Sobel(image, cv2.CV\_64F, 1, 0, ksize=3)

# Step 3: Apply Sobel filter for edge detection in the y-direction (Sobel-Y)

# Sobel filter is applied to the image to detect vertical edges. dx=0 and dy=1 indicates Sobel-Y.

sobel\_y = cv2.Sobel(image, cv2.CV\_64F, 0, 1, ksize=3)

# Step 4: Convert the Sobel filtered images to uint8 for display purposes

# Sobel results may contain negative values, so we use convertScaleAbs to convert the result to absolute values.

# This converts the result to an unsigned 8-bit integer (uint8) format for display.

sobel\_x = cv2.convertScaleAbs(sobel\_x)

sobel\_y = cv2.convertScaleAbs(sobel\_y)

# Step 5: Display the original and Sobel-filtered images using matplotlib

# The 'imshow' function of matplotlib displays the image. We use 'cmap="gray"' to show the image in grayscale.

plt.figure(figsize=(10, 7))

# Displaying the original image

plt.subplot(1, 3, 1), plt.imshow(image, cmap='gray'), plt.title('Original Image')

# Displaying the edge-detected image in the x-direction (Sobel-X)

plt.subplot(1, 3, 2), plt.imshow(sobel\_x, cmap='gray'), plt.title('Edge Detection - Sobel-X')

# Displaying the edge-detected image in the y-direction (Sobel-Y)

plt.subplot(1, 3, 3), plt.imshow(sobel\_y, cmap='gray'), plt.title('Edge Detection - Sobel-Y')

# Show the images

plt.show()

**OUTPUT:**

A screenshot of a computer screen

AI-generated content may be incorrect.

**Task 2: Implement Max Pooling and Average Pooling**

CODE:

import tensorflow as tf

import numpy as np

Step 1: Create a random 4x4 matrix (input image)

This matrix is a 4x4 matrix with random float values.

input\_matrix = np.array([[1, 2, 3, 4],

[5, 6, 7, 8],

[9, 10, 11, 12],

[13, 14, 15, 16]], dtype=np.float32)

# Step 2: Apply 2x2 Max Pooling

# MaxPooling2D performs max pooling over 2x2 regions with a stride of 2.

max\_pooling = tf.keras.layers.MaxPooling2D(pool\_size=(2, 2), strides=2, padding='valid')

# Step 3: Apply 2x2 Average Pooling

# AveragePooling2D performs average pooling over 2x2 regions with a stride of 2.

average\_pooling = tf.keras.layers.AveragePooling2D(pool\_size=(2, 2), strides=2, padding='valid')

# Apply AveragePooling to the input matrix, adding batch and channel dimensions

avg\_pooled\_output = average\_pooling(np.expand\_dims(np.expand\_dims(input\_matrix, axis=0), axis=-1))

# Step 4: Print the results

# Print the original matrix, the max-pooled output, and the average-pooled output.

print("Original Matrix:")

print(input\_matrix)

print("\nMax Pooled Matrix (2x2 Pooling):")

# .numpy() converts the tensor to a NumPy array and .squeeze() removes unnecessary dimensions.

print(max\_pooled\_output.numpy().squeeze())

print("\nAverage Pooled Matrix (2x2 Pooling):")

print(avg\_pooled\_output.numpy().squeeze())

**OUTPUT:**

A white background with black text

AI-generated content may be incorrect.

**4 ANS)**

**Task 1: Implement AlexNet Architecture**

CODE:

import tensorflow as tf

from tensorflow.keras import layers, models

def build\_alexnet():

model = models.Sequential()

# Conv2D Layer: 96 filters, kernel size = (11,11), stride = 4, activation = ReLU

model.add(layers.Conv2D(96, (11, 11), strides=4, activation='relu', input\_shape=(227, 227, 3)))

# MaxPooling Layer: pool size = (3,3), stride = 2

model.add(layers.MaxPooling2D(pool\_size=(3, 3), strides=2))

# Conv2D Layer: 256 filters, kernel size = (5,5), activation = ReLU

model.add(layers.Conv2D(256, (5, 5), activation='relu'))

# MaxPooling Layer: pool size = (3,3), stride = 2

model.add(layers.MaxPooling2D(pool\_size=(3, 3), strides=2))

# Conv2D Layer: 384 filters, kernel size = (3,3), activation = ReLU

model.add(layers.Conv2D(384, (3, 3), activation='relu'))

# Conv2D Layer: 384 filters, kernel size = (3,3), activation = ReLU

model.add(layers.Conv2D(384, (3, 3), activation='relu'))

# Conv2D Layer: 256 filters, kernel size = (3,3), activation = ReLU

model.add(layers.Conv2D(256, (3, 3), activation='relu'))

# MaxPooling Layer: pool size = (3,3), stride = 2

model.add(layers.MaxPooling2D(pool\_size=(3, 3), strides=2))

# Flatten Layer

model.add(layers.Flatten())

# Fully Connected (Dense) Layer: 4096 neurons, activation = ReLU

model.add(layers.Dense(4096, activation='relu'))

# Dropout Layer: 50%

model.add(layers.Dropout(0.5))

# Fully Connected (Dense) Layer: 4096 neurons, activation = ReLU

model.add(layers.Dense(4096, activation='relu'))

# Dropout Layer: 50%

model.add(layers.Dropout(0.5))

# Output Layer: 10 neurons, activation = Softmax

model.add(layers.Dense(10, activation='softmax'))

# Return the model

return model

# Build and compile the AlexNet model

alexnet\_model = build\_alexnet()

alexnet\_model.summary() # Display model summary

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**Task 2: Implement Residual Block and ResNet-like Model**

CODE:

def residual\_block(input\_tensor, filters):

"""

Implementing a residual block with two Conv2D layers

and a skip connection that adds the input to the output before activation.

"""

# First Conv2D layer with ReLU activation

x = layers.Conv2D(filters, (3, 3), activation='relu', padding='same')(input\_tensor)

# Second Conv2D layer with ReLU activation

x = layers.Conv2D(filters, (3, 3), activation='relu', padding='same')(x)

# Skip connection: Add the input tensor to the output of the second Conv2D

x = layers.add([x, input\_tensor])

return x

def build\_resnet():

"""

Build a simple ResNet-like model using residual blocks.

"""

input\_tensor = layers.Input(shape=(224, 224, 3)) # Input layer

# Initial Conv2D Layer

x = layers.Conv2D(64, (7, 7), strides=2, activation='relu', padding='same')(input\_tensor)

# First Residual Block

x = residual\_block(x, 64)

# Second Residual Block

x = residual\_block(x, 64)

# Flatten Layer

x = layers.Flatten()(x)

# Dense Layer: 128 neurons

x = layers.Dense(128, activation='relu')(x)

# Output Layer: Softmax activation for classification

output\_tensor = layers.Dense(10, activation='softmax')(x)

# Create the model

model = models.Model(inputs=input\_tensor, outputs=output\_tensor)

return model

# Build and compile the ResNet-like model

resnet\_model = build\_resnet()

resnet\_model.summary() # Display model summary

**OUTPUT:**

A screenshot of a computer program

AI-generated content may be incorrect.