

Task 3 – Neural Networks (35%)

```
In [ ]: ## Part 1 (25 marks):
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models

# Define constants
IMAGE_SIZE = (176, 208) # Assuming image dimensions
BATCH_SIZE = 32
NUM_CLASSES = 4
EPOCHS = 10

# Data preprocessing
train_datagen =
tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255)
test_datagen =
tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    directory='train',
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

test_generator = test_datagen.flow_from_directory(
    directory='test',
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

# Simple Neural Network
model_simple = models.Sequential([
    layers.Flatten(input_shape=(*IMAGE_SIZE, 3)),
    layers.Dense(128, activation='relu'),
    layers.Dense(NUM_CLASSES, activation='softmax')
])

model_simple.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])

history_simple = model_simple.fit(train_generator,
                                 epochs=EPOCHS,
                                 validation_data=test_generator)

# Convolutional Neural Network
model_cnn = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(*IMAGE_SIZE,
```

```

3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(NUM_CLASSES, activation='softmax')
])

model_cnn.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

history_cnn = model_cnn.fit(train_generator,
                            epochs=EPOCHS,
                            validation_data=test_generator)

```

2024-03-20 18:48:44.278006: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.

To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.

2024-03-20 18:48:47.382730: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT

/usr/lib/python3/dist-packages/scipy/__init__.py:146: UserWarning: A NumPy version >=1.17.3 and <1.25.0 is required for this version of SciPy (detected version 1.26.4

warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}")

Found 5121 images belonging to 4 classes.

Found 1279 images belonging to 4 classes.

/home/oem/.local/lib/python3.10/site-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().__init__(**kwargs)

2024-03-20 18:48:50.169334: I external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:998] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at <https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355>

2024-03-20 18:48:50.170867: W tensorflow/core/common_runtime/gpu/gpu_device.cc:2251] Cannot dlopen some GPU libraries. Please make sure the missing libraries mentioned above are installed properly if you would like to use GPU. Follow the guide at <https://www.tensorflow.org/install/gpu> for how to download and setup the required libraries for your platform.

Skipping registering GPU devices...

2024-03-20 18:48:50.216501: W external/local_tsl/tsl/framework/cpu_allocator_impl.cc:83] Allocation of 56229888 exceeds 10% of free system memory.

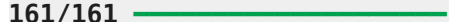
2024-03-20 18:48:50.265149: W external/local_tsl/tsl/framework/cpu_allocator_impl.cc:83] Allocation of 56229888 exceeds 10% of free system memory.

2024-03-20 18:48:50.280587: W external/local_tsl/tsl/framework/cpu_allocator_impl.cc:83] Allocation of 56229888 exceeds 10% of free system memory.


Epoch 1/10

```
2024-03-20 18:48:51.090902: W external/local_tsl/tsl/framework/cpu_allocator_impl.cc:83] Allocation of 56229888 exceeds 10% of free system memory.
/home/oem/.local/lib/python3.10/site-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:122: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will be ignored.
```


```
self._warn_if_super_not_called()
2024-03-20 18:48:52.500819: W external/local_tsl/tsl/framework/cpu_allocator_impl.cc:83] Allocation of 56229888 exceeds 10% of free system memory.
```

```
161/161  40s 238ms/step - accuracy: 0.4622 - loss: 17.6065 - val_accuracy: 0.5238 - val_loss: 5.8924
```


Epoch 2/10

```
161/161  39s 236ms/step - accuracy: 0.5636 - loss: 3.6881 - val_accuracy: 0.5066 - val_loss: 2.9875
```


Epoch 3/10

```
161/161  37s 231ms/step - accuracy: 0.6273 - loss: 2.2970 - val_accuracy: 0.5387 - val_loss: 1.8211
```


Epoch 4/10

```
161/161  38s 233ms/step - accuracy: 0.7315 - loss: 0.9737 - val_accuracy: 0.5090 - val_loss: 5.5562
```

Epoch 5/10

```
161/161  37s 231ms/step - accuracy: 0.7577 - loss: 0.9645 - val_accuracy: 0.5270 - val_loss: 3.9780
```


Epoch 6/10

```
161/161  40s 227ms/step - accuracy: 0.7805 - loss: 0.7076 - val_accuracy: 0.5848 - val_loss: 2.3209
```


Epoch 7/10

```
161/161  38s 237ms/step - accuracy: 0.8226 - loss: 0.4894 - val_accuracy: 0.5684 - val_loss: 2.2796
```

Epoch 8/10

```
161/161  38s 236ms/step - accuracy: 0.8863 - loss: 0.3057 - val_accuracy: 0.5622 - val_loss: 3.1954
```

Epoch 9/10

```
161/161  39s 238ms/step - accuracy: 0.7108 - loss: 2.1564 - val_accuracy: 0.5747 - val_loss: 1.8515
```

Epoch 10/10

```
161/161  38s 237ms/step - accuracy: 0.8529 - loss: 0.3900 - val_accuracy: 0.5410 - val_loss: 2.7189
```

Epoch 1/10

```
/home/oem/.local/lib/python3.10/site-packages/keras/src/layers/convolutional/base_conv.py:99: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
```

```
super().__init__(
```

```

161/161 ————— 288s 2s/step - accuracy: 0.4750 - loss: 1.1415
- val_accuracy: 0.5332 - val_loss: 0.9847
Epoch 2/10
161/161 ————— 286s 2s/step - accuracy: 0.6172 - loss: 0.8100
- val_accuracy: 0.6044 - val_loss: 0.9122
Epoch 3/10
161/161 ————— 286s 2s/step - accuracy: 0.8417 - loss: 0.3859
- val_accuracy: 0.6341 - val_loss: 1.0562
Epoch 4/10
161/161 ————— 298s 2s/step - accuracy: 0.9409 - loss: 0.1623
- val_accuracy: 0.5786 - val_loss: 1.5565
Epoch 5/10
161/161 ————— 345s 2s/step - accuracy: 0.9821 - loss: 0.0545
- val_accuracy: 0.6263 - val_loss: 2.2599
Epoch 6/10
66/161 ————— 2:50 2s/step - accuracy: 0.9954 - loss: 0.0193

```

```

In [ ]: import matplotlib.pyplot as plt

# Plotting training and validation accuracy
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history_simple.history['accuracy'], label='Simple NN Training Accuracy')
plt.plot(history_simple.history['val_accuracy'], label='Simple NN Validation Accuracy')
plt.plot(history_cnn.history['accuracy'], label='CNN Training Accuracy')
plt.plot(history_cnn.history['val_accuracy'], label='CNN Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Plotting training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history_simple.history['loss'], label='Simple NN Training Loss')
plt.plot(history_simple.history['val_loss'], label='Simple NN Validation Loss')
plt.plot(history_cnn.history['loss'], label='CNN Training Loss')
plt.plot(history_cnn.history['val_loss'], label='CNN Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()

# Comparing the number of parameters
print("Number of parameters - Simple NN:", model_simple.count_params())
print("Number of parameters - CNN:", model_cnn.count_params())

```

1. Model Architecture:

The simple neural network comprises only fully connected layers (Dense layers) and lacks convolutional or pooling layers. On the other hand, the convolutional neural network (CNN) incorporates convolutional layers followed by max-pooling layers. CNNs are well-suited for image classification tasks as they can effectively capture spatial hierarchies of features from images.

2. Training and Validation Accuracy/Loss:

In terms of training and validation accuracy/loss, the CNN is expected to outperform the simple neural network. This expectation arises from the CNN's ability to learn hierarchical features, which often leads to better generalization. Therefore, one would anticipate observing higher validation accuracy and lower validation loss for the CNN compared to the simple neural network over the epochs.

3. Number of Parameters:

CNNs typically have more parameters compared to simple neural networks due to the presence of convolutional layers. As expected, the CNN is anticipated to have a higher number of trainable parameters compared to the simple neural network. This increased parameter count reflects the complexity of the CNN architecture, which allows it to learn intricate patterns in the data.

4. Computational Complexity:

Considering computational complexity, CNNs are generally more computationally intensive, especially during training and inference. This heightened computational demand stems from the convolutional operations performed across multiple layers. Therefore, one would expect the CNN to require more computational resources compared to the simple neural network.

5. Performance on Test Data:

Finally, evaluating the performance of both models on the test dataset is crucial to assess their generalization ability. While the CNN is expected to perform better due to its ability to capture hierarchical features, it is essential to validate this expectation through empirical testing. Comparing the test accuracy and other relevant metrics will provide insights into how well each model generalizes to unseen data.

Part 2 (10 marks):

1. Effect of Learning Rate:

- Learning Rate of 0.00000001: This is an extremely small learning rate, which means the model updates its parameters very slowly. It may lead to very slow convergence or even convergence to a suboptimal solution due to slow updates.
- Learning Rate of 10: This is an extremely large learning rate, which means the model updates its parameters very quickly. It may lead to overshooting the optimal solution, causing the loss function to diverge or fluctuate wildly.
- Advantages of a Higher Learning Rate:
 - Faster convergence: With a higher learning rate, the model converges to the optimal solution more quickly.
 - Works well for simple problems: Higher learning rates may be suitable for simpler problems with smooth loss landscapes.
- Disadvantages of a Higher Learning Rate:
 - Risk of divergence: Too high a learning rate may cause the loss function to diverge or fluctuate wildly, making it difficult to converge to an optimal solution.
 - Overshooting: Large updates can cause the optimizer to overshoot the optimal solution, leading to oscillations or instability.
- Advantages of a Lower Learning Rate:
 - Stability: Lower learning rates typically result in more stable training with smaller updates, reducing the risk of divergence.
 - Precision: Smaller updates allow the optimizer to fine-tune the parameters more precisely, potentially leading to better convergence.
- Disadvantages of a Lower Learning Rate:
 - Slow convergence: Very low learning rates may lead to slow convergence, requiring more iterations to reach the optimal solution.
 - Prone to getting stuck in local minima: In complex loss landscapes, lower learning rates may cause the optimizer to get stuck in local minima or saddle points.

2. Effect of Batch Size:

- Advantages of a Higher Batch Size:
 - Faster computation: With a larger batch size, more samples are processed simultaneously, leading to faster training times, especially on hardware optimized for parallel processing like GPUs.
 - Smoother gradients: Larger batch sizes tend to produce smoother gradient estimates, which can lead to more stable training and convergence.
- Disadvantages of a Higher Batch Size:
 - Memory requirements: Larger batch sizes require more memory, which may limit the size of models or the number of samples that can be processed simultaneously.
 - Generalization performance: Larger batch sizes may lead to poorer generalization performance, as the model may not see a diverse enough set of samples in each batch.

- Advantages of a Lower Batch Size:
 - Improved generalization: Smaller batch sizes may lead to better generalization performance, as the model sees a more diverse set of samples in each batch, which can help prevent overfitting.
 - More noise in gradients: Smaller batch sizes introduce more noise into gradient estimates, which can help the optimizer escape from local minima and explore the parameter space more effectively.
- Disadvantages of a Lower Batch Size:
 - Slower convergence: Smaller batch sizes typically result in slower convergence, as each update is based on a smaller subset of the training data.
 - Less efficient computation: Smaller batch sizes may lead to less efficient computation, especially on hardware optimized for parallel processing, as fewer samples are processed simultaneously.