THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Lab Demonstration 2
Pattern Recognition

Shekhar “Shakes” Chandra

Version 1.66

In this lab, we will study dimensionality reduction, classification and deep learning pipelines as a way of learning the basics of artificial intelligence (AI) frameworks such as Tensorflow (TF)/PyTorch. The foundation of these frameworks is based on linear algebra in a similar manner to Numpy. As a consequence, a number of functions from Numpy are also available in these frameworks.

The laboratory is divided into four main parts. Firstly, we will have a short exercise to ensure that the prerequisite knowledge from the previous laboratory is well understood before moving on to the new content, while going over the recently covered course material on the discrete Fourier transform (DFT). Next, we will look to introduce principal component analysis (PCA) of human faces and classification using the random forest, where you will be asked to create a TF/PyTorch equivalent model as your first deep learning model. Then build a traditional convolutional neural network (CNN) approach to do the same. Finally, you will have the opportunity to implement a CNN solution of your choice from a list of problems and data to solve an example recognition problem.

Use of Artificial Intelligence

This lab is designed to incorporate the use of AI models to assist in your learning experiences. Models such as GitHub Co-Pilot (available for free to students), Microsoft Co-Pilot, Google Gemini, OpenAI ChatGPT (or equivalent) or Claude will be permitted as described in the instructions below. The goal is to show how these models can be useful in understanding and improving your learning and workflows, but to understand their limitations and common problems with their use as well.

Demonstration

This is the lab sheet for the Pattern Recognition Demonstration. Marks achievable are indicated for each section. You will be awarded marks based on the completed tasks (ideally all tasks) AND demonstrate it to the instructor in one of your practical sessions BEFORE the due date. This includes the ability to answer questions related to the tasks and its content. Please check the course instance page for the correct due date.

1 Part 1 of 4 - Discrete Fourier Transform (1 Mark)

In linear algebra, a vector can be represented in different bases. Similarly, in signal processing, a signal can be expressed in either the time domain (as impulses at each instant) or the frequency domain (as sine and cosine waves). The Fourier transform is the operation that converts a signal from the time domain to the frequency domain, revealing how much of each frequency component is present. A square wave offers a particularly insightful example: In the time domain, it appears simple, with sharp edges, yet in the frequency domain, its Fourier transform reveals an infinite series of odd harmonics of the fundamental frequency. In the following section, we examine how these harmonics can be combined to reconstruct a square wave. Run the following code (also found in the this notebook) and plot the square wave and the provided harmonics.

```
import numpy as np
import matplotlib.pyplot as plt
import time

# Set parameters for the signal
N = 2048                    # Number of sample points
T = 1.0                     # Duration of the signal in seconds
f0 = 1                      # Fundamental frequency of the square wave in Hz

# List of harmonic numbers used to construct the square wave
harmonics = [1, 3, 5]

# Define the square wave function
def square_wave(t):
    return np.sign(np.sin(2.0 * np.pi * f0 * t))

# Fourier series approximation of the square wave
def square_wave_fourier(t, f0, N):
    result = np.zeros_like(t)
    for k in range(N):
        # The Fourier series of a square wave contains only odd harmonics.
        n = 2 * k + 1
        # Add harmonics to reconstruct the square wave.
        result += np.sin(2 * np.pi * n * f0 * t) / n
    return (4 / np.pi) * result

# Create the time vector
# np.linspace generates evenly spaced numbers over a specified interval.
# We use endpoint=False because the interval is periodic.
t = np.linspace(0.0, T, N, endpoint=False)

# Generate the original square wave
square = square_wave(t)
```

```
    plt.figure(figsize=(12, 8))
    # Plot the original square wave
    plt.subplot(2, 3, 1)
    plt.plot(t, square, 'k', label="Square wave")
    plt.title("Original Square Wave")
    plt.ylim(-1.5, 1.5)
    plt.grid(True)
    plt.legend()
    # Plot Fourier reconstructions under different number of harmonics
    for i, Nh in enumerate(harmonics, start=2):
        plt.subplot(2, 3, i)
        y = square_wave_fourier(t, f0, Nh)
        plt.plot(t, y, label=f"N={Nh} harmonics")
        plt.plot(t, square, 'k--', alpha=0.5, label="Square wave")
        plt.title(f"Fourier Approximation with N={Nh}")
        plt.ylim(-1.5, 1.5)
        plt.grid(True)
        plt.legend()

    plt.tight_layout()
    plt.show()
```

Now that you have successfully reconstructed a square wave, try increasing the number of harmonics used in the reconstruction and consider the effect of adding higher-order harmonics, such as 20 or 50. How do these additional components influence the accuracy and sharpness of the square wave? Furthermore, once the square wave has been constructed, we can take a step further by decomposing it back into its harmonic components in the Fourier domain using the DFT algorithm. To illustrate this process, the teaching team has generated code—using both the naive and fast algorithms for computing the DFT—with the assistance of Google Gemini (2.5 Pro, also available for free) that can be found here. Modify this code to apply it to the square wave or use the code shown below to compare the frequency components obtained from the DFT with those originally used to construct the square wave. Do you notice any differences, and if so, why might they occur?

```
# 2. Apply the DFT and time the execution

def naive_dft(x):
    """
    Compute the Discrete Fourier Transform (DFT) of a 1D signal.

    This is a "naïve" implementation that directly follows the DFT formula,
    which has a time complexity of O(N^2).

    Args:
        x (np.ndarray): The input signal, a 1D NumPy array.
    Returns:
        np.ndarray: The complex-valued DFT of the input signal.
    """
    N = len(x)
    # Create an empty array of complex numbers to store the DFT results
    X = np.zeros(N, dtype=np.complex128)

    # Iterate through each frequency bin (k)
    for k in range(N):
        # For each frequency bin, sum the contributions from all input samples (n)
        for n in range(N):
            # The core DFT formula: x[n] * e^(-2j * pi * k * n / N)
            angle = -2j * np.pi * k * n / N
            X[k] += x[n] * np.exp(angle)

    return X

# Construct a square wave using 50 harmonics
signal = square_wave_fourier(t, f0, 50)
# Time the naïve DFT implementation
start_time_naive = time.time()
dft_result = naive_dft(signal)
end_time_naive = time.time()
naive_duration = end_time_naive - start_time_naive

# Time NumPy's FFT implementation
start_time_fft = time.time()
fft_result = np.fft.fft(signal)
end_time_fft = time.time()
fft_duration = end_time_fft - start_time_fft

# 3. Print Timings and Verification
print("--- DFT/FFT Performance Comparison ---")
print(f"Naïve DFT Execution Time: {naive_duration:.6f} seconds")
print(f"NumPy FFT Execution Time: {fft_duration:.6f} seconds")
# It's possible for the FFT to be so fast that the duration is 0.0, so we handle that case.
if fft_duration > 0:
    print(f"FFT is approximately {naive_duration / fft_duration:.2f} times faster.")
else:
    print("FFT was too fast to measure a significant duration difference.")

# Check if our implementation is close to NumPy's result
# np.allclose is used for comparing floating-point arrays.
print(f"\nOur DFT implementation is close to NumPy's FFT: {np.allclose(dft_result, fft_result)
```

```python
# 4. Prepare for Plotting
# Generate the frequency axis for the plot.
# np.fft.fftfreq returns the DFT sample frequencies.
# We only need the first half of the frequencies (the positive ones) due to symmetry.
xf = np.fft.fftfreq(N, d=T/N)[:N//2]
# We normalize the magnitude by N and multiply by 2 to get the correct amplitude.
magnitude = 2.0/N * np.abs(dft_result[0:N//2])

# 5. Visualize the Results
plt.style.use('seaborn-v0_8-darkgrid')
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10))

# Plot the original time-domain signal
ax1.plot(t, signal, color='c')
ax1.set_title('Input Sine Wave Signal', fontsize=16)
ax1.set_xlabel('Time (s)', fontsize=12)
ax1.set_ylabel('Amplitude', fontsize=12)
ax1.set_xlim(0, 1.0) # Show a few cycles of the sine wave
ax1.grid(True)

# Plot the frequency-domain signal (magnitude of the DFT)
ax2.stem(xf, magnitude, basefmt=" ")
ax2.set_title(
    'Discrete Fourier Transform (Magnitude Spectrum)',
    fontsize=16
    )
ax2.set_xlabel('Frequency (Hz)', fontsize=12)
ax2.set_ylabel('Magnitude', fontsize=12)
ax2.set_xlim(0, 50) # Focus on lower frequencies
ax2.grid(True)

# Add vertical lines for the first ten frequencies
for i in range(20):
    if i < len(xf) and i % 2 == 1:  # Only plot odd harmonics
        ax2.axvline(
            xf[i], color='r', linestyle='--', alpha=0.7,
            label=f'f{i}: {i}* f0 = {xf[i]:.1f} Hz'
            )

# Only show labels for first 3 frequencies to avoid cluttering
ax2.legend()
```

```
plt.tight_layout()
plt.show()
```

With an AI of you choice or by hand using this code provided, you should be able to (re)use the code in a script or notebook locally or re-generate it using AI because you will need to modify the code in the next step. Now modify the 'square_wave', 'square_wave_fourier' and 'naive_dft' functions so that they are implemented using TensorFlow (TF) or PyTorch operations. For 'naive_dft' in particular, create a second version that explicitly runs on the GPU using TF/PyTorch tensor operations (rather than their built-in FFT functions, which behave similarly to NumPy's implementation). Compare its computation time with the other two methods, note the times in order of fastest to slowest. Change the size of the data and note the change in timings of the three methods. Can you explain the order of the methods in terms of time, i.e., why is the fastest method the fastest you have observed?

2 Part 2 of 4 - Eigenfaces (2 Marks)

We will compute Eigenfaces - the PCA of human faces using Numpy and the funnelled "Labeled Faces in the Wild" (LFW). Scikit learn Eigenfaces example is modified below to show how the PCA model of faces is constructed using Numpy arrays.

First load the relevant data and functions

```
from sklearn.datasets import fetch_lfw_people
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import numpy as np

# Download the data, if not already on disk and load it as numpy arrays
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
```

Extract the meaningful parameters of the faces dataset

```
# introspect the images arrays to find the shapes (for plotting)
n_samples, h, w = lfw_people.images.shape

# for machine learning we use the 2 data directly (as relative pixel
# positions info is ignored by this model)
X = lfw_people.data
n_features = X.shape[1]

# the label to predict is the id of the person
y = lfw_people.target
target_names = lfw_people.target_names
n_classes = target_names.shape[0]

print("Total dataset size:")
print("n_samples: %d" % n_samples)
print("n_features: %d" % n_features)
print("n_classes: %d" % n_classes)
```

It is important in machine learning to split the data accordingly into training and testing sets to avoid contamination of the model. Ideally, you should also have a validation set.

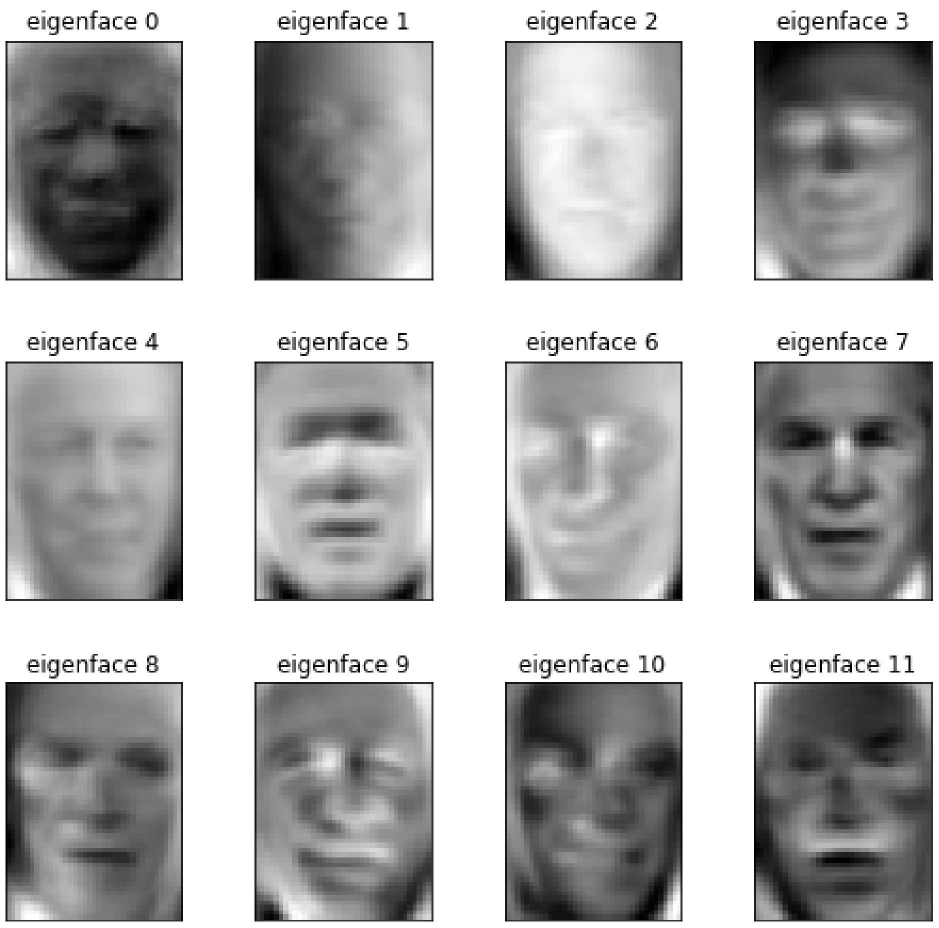
```
# Split into a training set and a test set using a stratified k fold
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)

# Compute a PCA (eigenfaces) on the face dataset (treated as unlabeled
# dataset): unsupervised feature extraction / dimensionality reduction
n_components = 150
```

Compute the PCA via eigen-decomposition of the data matrix X after the mean of the training set is removed. This results in a model with variations from the mean. We also transform the training and testing data into 'face space', i.e. the learned sub space of the eigen-faces.

```
# Center data
mean = np.mean(X_train, axis=0)
X_train -= mean
X_test -= mean

#Eigen-decomposition
U, S, V = np.linalg.svd(X_train, full_matrices=False)
components = V[:n_components]
eigenfaces = components.reshape((n_components, h, w))

#project into PCA subspace
X_transformed = np.dot(X_train, components.T)
print(X_transformed.shape)
X_test_transformed = np.dot(X_test, components.T)
print(X_test_transformed.shape)
```

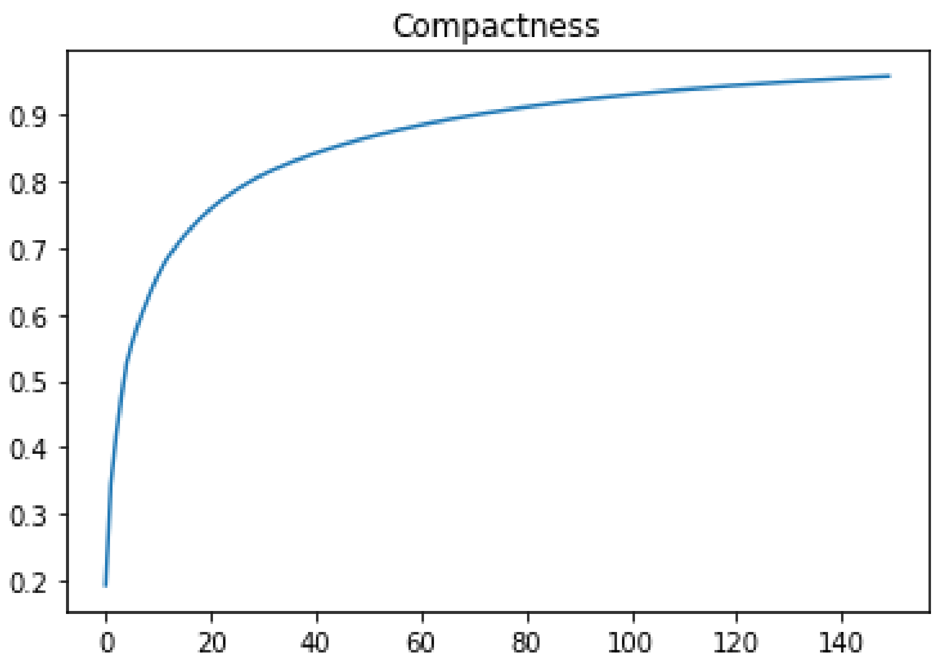
Finally, plot the resulting eigen-vectors of the face PCA model, AKA the eigenfaces

```
import matplotlib.pyplot as plt

# Qualitative evaluation of the predictions using matplotlib
def plot_gallery(images, titles, h, w, n_row=3, n_col=4):
    """Helper function to plot a gallery of portraits"""
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())

eigenface_titles = ["eigenface %d" % i for i in range(eigenfaces.shape[0])]
plot_gallery(eigenfaces, eigenface_titles, h, w)

plt.show()
```



We should always evaluate the performance of the dimensionality reduction via a compactness plot

```
explained_variance = (S ** 2) / (n_samples - 1)
total_var = explained_variance.sum()
explained_variance_ratio = explained_variance / total_var
ratio_cumsum = np.cumsum(explained_variance_ratio)
print(ratio_cumsum.shape)
eigenvalueCount = np.arange(n_components)

plt.plot(eigenvalueCount, ratio_cumsum[:n_components])
plt.title('Compactness')
plt.show()
```



Use the PCA 'face space' as features and build a random forest classifier to classify the faces according to the labels. We then view its classification performance.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

#build random forest
estimator = RandomForestClassifier(n_estimators=150, max_depth=15, max_features=150)
estimator.fit(X_transformed, y_train) #expects X as [n_samples, n_features]

predictions = estimator.predict(X_test_transformed)
correct = predictions==y_test
total_test = len(X_test_transformed)
#print("Gnd Truth:", y_test)
print("Total Testing", total_test)
print("Predictions", predictions)
print("Which Correct:",correct)
print("Total Correct:",np.sum(correct))
print("Accuracy:",np.sum(correct)/total_test)

print(classification_report(y_test, predictions, target_names=target_names))
```

This should show you the performance of the RF/PCA based face recognition. (1 Mark)

Your task for this part is to re-implement the above PCA algorithm of the Eigenfaces problem using TF or PyTorch functions. (2 Marks)

3 Part 3 of 4 - CNNs (7 Marks)

In this part, you will create a CNN based classifier that should hopefully out perform the above Eigenfaces algorithm from the previous part. Multiple layers will be strung together and tied to a fully connected (or dense) layers to result in a classified output.

3.1 CNN Classifier (1 Mark)

Using either TF/Keras/PyTorch/JAX, implement a CNN based classifier for the same LFW dataset from Part 1 with two convolution layers of 3x3 with 32 filters each and dense layers for classification.

You can reuse the initial elements of your code in Part 1 that loads and creates the training and the testing sets. You may use the Adam optimiser and (sparse) categorical cross entropy loss. Note that convolution layers expect 4D tensors so that you will need normalise and resize your arrays, see for example

```
# The following code could be used for data preprocessing if using pytorch
X = lfw_people.images
Y = lfw_people.target
# Verify the value range of X_train. No normalization is necessary in this case,
#  as the input values already fall within the range of 0.0 to 1.0.
print("X_min:",X.min(),"X_train_max:", X.max())
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25, random_state=42)
X_train = X_train[:, np.newaxis, :, :]
X_test = X_test[:, np.newaxis, :, :]
print("X_train shape:", X_train.shape)
```

3.2 Advanced Git Course (2 Marks)

For the next part of this section, please complete the second Git Short Course: Version Control for Teams using Git. Register with edX in order to access the short course if you haven't already done so. If you do not already have an edX account linked to your UQ student account, you will need to create an account. Additional instructions on accessing this short course (including most up-to-date links, as these change every semester) will be provided as a Blackboard or Eds Discussion board post.

3.3 DAWNBench Challenge (4 Marks)

Finally, construct a Fast CIFAR10 dataset classification network using one of TF/Keras/PyTorch/JAX to meet the standards of the DAWNBench challenge as the following:

1. The model must achieve more than 93% accuracy and that is trainable in the fastest time possible (usually under 30 mins on a cluster).
2. Usage of pre-built models will generally not be allowed unless approved by the demonstrator.
3. The model must run inference or a single epoch of training on the Ranpur compute cluster (see appendix A for details) during the demonstration.

Note that using the Ranpur cluster will be necessary for all BUT the lowest difficulty (and lowest marks) for the next part of this assessment.

Using a ResNet-18 and mixed precision, it is possible to achieve 94% on a model trained for only approximately 360 seconds on a NVIDIA V100 GPU on a cluster. You have access to the newer NVIDIA A100 GPU on UQ's Rangpur cluster. Can you achieve a time that is equivalent or faster? See appendix B for more resources on the DAWNBench challenge, including code that you can port.

4 Part 4 of 4 - Recognition (10 Marks)

For this part, you will need to develop your own solution to the recognition problems provided below. There are a number of problems available, each having different levels of difficulty and will ultimately affect the total maximum mark attainable for this section. Partial marks can be awarded (with respect to the maximum) for having a go at crafting a solution to any of the problems below and answering some of the questions.

4.1 Requirements

Solve the following problems in the subsequent sections using TF/Keras/PyTorch, noting that the marks are allocated according to the level of difficulty and to obtain all marks you will need to solve **ALL** of the tasks. You must obtain reasonable results and be able to explain all results, network layers and code to the demonstrator in order to obtain the marks available. Usage of pre-built/pre-trained models will generally not be allowed unless approved by the demonstrator.

[You must create a GitHub project in your own account for the tasks below with relevant commit logs in addition to your demonstration to receive ANY marks for this part of the lab. The demonstrator may request evidence that this is your own account by logging into the account during the demo. Note that it is okay to refer to web sources for learning purposes and to use AI to help. However, the teaching team may not be able to assist with issues encountered from sources not supported by the teaching team or erroneous code generated by AI. Please cite the sources appropriately, and the instructors will check your understanding thoroughly with questions they feel is appropriate. You must convince and justify your code to the satisfaction of your instructor.]

4.2 Marking Criteria

Marks will be awarded according to the following breakdown:

1. Code functions as required and completes the tasks. (1-6 Marks based on tasks completed)
2. GitHub project is valid and hosts code correctly (1 Marks)
3. Code is commented and structured correctly for use by others (1 Marks)
4. Commit messages are reasonable and meaningful (1 Marks)

4.3 Tasks

IMPORTANT: The following recognition tasks are progressively more difficult and marks are awarded accordingly. We recommend that everyone attempt at least the easy task below. However, the normal and hard difficulties are recommended only for those who wish to be deep learning and AI experts as solving each of these tasks below takes as much time as all previous tasks in this sheet combined!

The Link to the Next Assessment

The next assessment is the report/project and it will include some recognition tasks that will be built upon on the tasks below. So learning and completing as many of the tasks below will set you up the best possible way for the report/project assessment later in the course.

4.3.1 Variational Autoencoder - Easy (VAE: Maximum 4 Marks out of 10)

Construct a Variational Autoencoder (VAE) of the magnetic resonance (MR) images of the brain via the Preprocessed OASIS dataset, which is found in the /home/groups/comp3710/ directory on the Rangpur cluster.

To obtain full marks, you might train the model and visualise the resulting manifold created by the VAE. For the visualisation, you may use sampling methods to create a 2D image of the manifold or use a dimensionality reduction technique such as UMAP.

4.3.2 UNet - Medium (VAE+UNet: Maximum 8 Marks out of 10)

UNet based magnetic resonance (MR) image segmentation of the brain via the Preprocessed OASIS dataset, which is found in the /home/groups/comp3710/ directory on the Rangpur cluster.

The segmentation accuracy of your model will need to be validated and achieve > 0.9 DSC for all labels. You must use categorical (one-hot) output in your network. Not doing so (i.e. without one-hot/categorical segmentations) may be a small deduction. Accuracy as mentioned must be obtained in either case. You must also visualise some of the segmentation results to justify the DSC scores obtained. You must run inference at demonstration on a test set and show the model is working correctly during the demo.

A note of warning, this requires knowledge of segmentation and skip connections within deep learning. You must explain each element of your training, the methods used to obtain segmentation model and justify the accuracies obtained. You will have to demonstrate the inference of the model via segmentation of an MRI from the dataset during the demo!

4.3.3 Generative Adversarial Networks - Hard (VAE+UNet+GAN: Maximum 10 Marks out of 10)

Realistic brain generation using generative adversarial networks (GANs) of the Preprocessed OASIS dataset, which is found in the /home/groups/comp3710/ directory on the Rangpur cluster.

Images must be suitably realistic and evidence of training (generated images, training loss plots etc.) of the models must be provided. The 'realism' of the brains will be judged by your instructor, so check with them if the results are appropriate for full or partial marks.

A note of warning, GANs have very chaotic convergence and therefore difficult to train. Only attempt it if you feel confident with deep learning and if you do, you do so at your own risk. You might want to start out using the MNIST digit dataset or the CelebA dataset first for GANs before attempting it with the OASIS dataset. Results obtained must look reasonably like brains for full marks. Full marks will only be awarded for OASIS results for GANs, partial marks for other results.

5 Appendix

A Rangpur High Performance Computing Cluster

Information on connecting and accessing the Ranpur computing cluster can be found here (login required). You can see an introduction to using such computing clusters and the SLURM queuing system in the HPC Summer of AI 2022 video. The OASIS dataset can be found in the /home/groups/comp3710/ directory on the Rangpur cluster.

B DAWNBench Resources

There a number of resources required that you can use to accomplish the DAWNBench challenge task:

- You can find the Crash Course in Deep Learning Summer of AI 2022 video that walks you through a DAWNBench solution.
- See also Shakes' JAX vision library code that solves the challenge here.