CNN for Early Alzheimer Disease Detection

1. Research Proposal

Alzheimer Disease (AD) is a progressive neurodegenerative disease that may lead to memory loss as a result of damaging the brain cells, which may also progress to a characteristic dementia syndrome. As there is currently no cure for AD, early detection of this disease is crucial to aid the potential patient to find a proper treatment to slow down the degenerative progress, reduce the symptoms in the short term and therefore alleviate the burden on the family and caregiver. Therefore, the aim of this project is to develop a model that uses Convolutional Neural Network (CNN) to detect images and predict if the patient is diagnosed with AD.

According to Guerchet, M. et. al. (2020) in their study, as of 2015, approximately 50 millions people around the world are diagnosed with AD and this figure is expected to keep rising in the future.

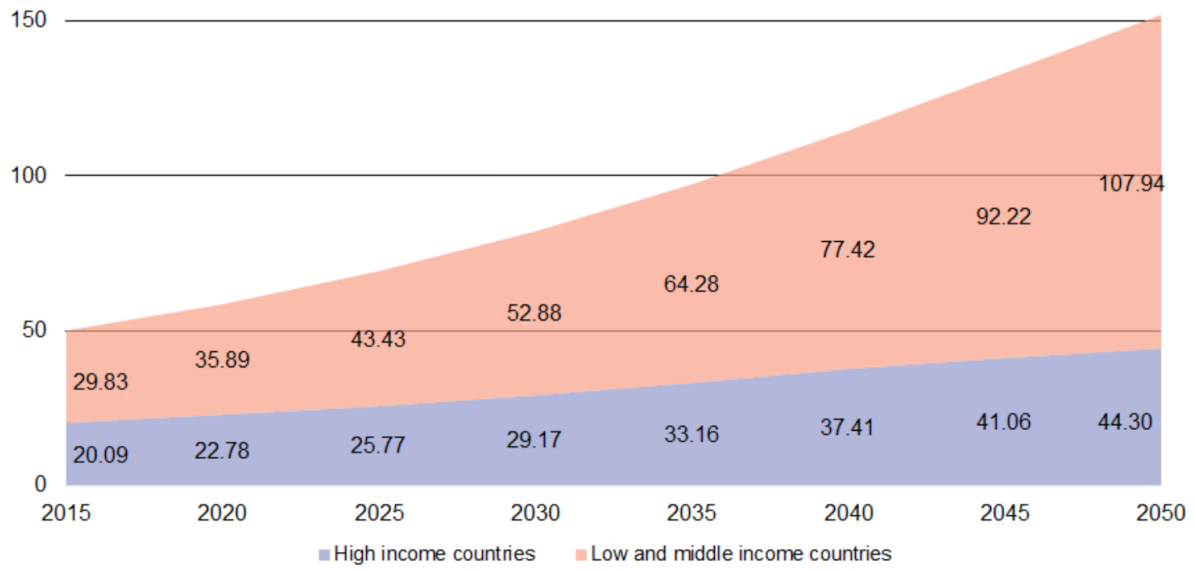


Figure 1. *Population diagnosed with AD (Guerchet, M. et. al., 2020)*

Furthermore, AD does not only affect the patient but also their family members as it will take a toll on their physical and mental health. The effect is not only restricted to these as it will also be financially challenging for the family members to support the patient. Grabher, B. J. (2018) in her study states that in the US, caregivers provide AD patients approximately 18.4 billion hours of unpaid care and their family members have to spend more than $10,000 a year to support the treatment. Overall, AD cost the US nearly US$277 billion. This means that they will have less disposable income left to be spent on the daily necessities and opportunity costs might arise from the increase in government's spending on supporting the patients should they rely on the government programs.

Parameter	2013	2015	2016
Number of family and other unpaid caregivers	15.5 million	15.9 million	15.9 million
Hours of unpaid care	17.7 billion	18.1 billion	18.2 billion
Hours of care per caregiver per week	21.9 h	21.9 h	21.9 h
Hours of care per caregiver per year	1,139 h	1,139 h	1,139 h
Hourly rate	$12.45	$12.25	$12.65
Total cost	$220.3 billion	$221.3 billion	$230 billion

***Figure 2**. Cost of treating AD (Grabher, B. J., 2018)*

In terms of mental health, the study also shows that taking care of AD patients induce some form of stress, such as work-related, time-related, physical and emotional stress. This is because some of the caregivers will have to reschedule their working hours to take care of the patient, while some others are juggling their work and caregiving role while at the same time facing financial issues to provide for adequate service.

Therefore, by being able to detect the sign of AD early, patients and caregivers will then be able to prepare for their next decision to treat the disease before it progresses beyond the stage where treatments are effective. Barber R. C. (2010) states that some benefits of detecting AD early include a lower treatment bill, ability to predict the rate at which AD progresses, possibility of being enrolled in clinical trials and impose less stress on the caregivers and family members as the patient will be able to attain the required attention in the shortest possible time.

Furthermore, early detection of AD will also alleviate some financial burden on the national budget in countries where the country provides medical benefits such as the US. Study conducted by Leifer, B. P. (2003) shows that the yearly cost savings associated with AD treatment such as prescription drugs and consultation fee amount to $3,891 for each patient. Furthermore, this reduces the need for a caregiver as the symptoms of AD could be mitigated early and put less stress on the family member.

Hence, with this in mind, several studies have tried to use imaging techniques to detect the possibility of AD occurring. Salehi A. W., et. al. (2020) uses CNN for early diagnosis with the sample Magnetic Resonance Imaging (MRI) images. This is because they believe that the use of Deep Learning has mostly been implemented for “automatic diagnosis” of AD to “satisfy clinicians primary goals.” Their result also shows a high accuracy ranging between 89 to 99 percent. Therefore, in our study, CNN will be implemented to validate if it is indeed reliable enough to predict AD.

2. Data

The dataset used is Alzheimer MRI Preprocessed Dataset (https://www.kaggle.com/datasets/sachinkumar413/alzheimer-mri-dataset) which is adapted from Kaggle. It consists of 6400 images, in which all of them belong to one of the 4 classes - namely Non- Demented, Very Mild Demented, Mild Demented and Moderate Demented. Each of the original images are resized to 128 by 128 pixels. Below are the sample images from each class.

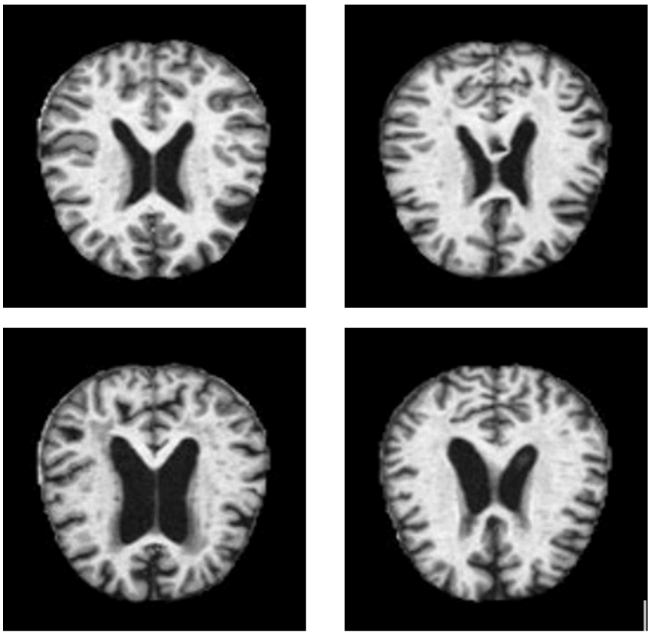


Figure 3. *Brain MRI for Non-Demented (Top Left), Very Mild Demented (Top Right), Mild Demented (Bottom Left) and Moderate Demented (Bottom Right)*

The dataset is resized to 256 by 256 pixels to increase the trainable parameters on the model. However, one drawback is that it will increase the computation time. Since the dataset is considered medium size, hence will be able to produce the model result in reasonable time. The dataset will then be split to training set and testing set (train_data and val_data) with the ratio of 80 to 20. There is also no missing data since the dataset are images, hence no data cleaning method is required.

Furthermore, another form of data pre-processing is also applied on the transfer learning stage as seen in Section 6 as the build-in model are not able to run using the current method of data splitting technique. However, the same dataset and splitting ratio are still used, hence there should only be marginal difference, which does not affect the performance of the model significantly.

3. Modelling

CNN structure is adopted in the model, in which the algorithm will take in the images as inputs and assign weights to the objects and hence be able to differentiate one image from the other. Therefore, with sufficient training, it will be able to learn the characteristics. In our model, we adopted Sequential class, as it groups several layers inside into the model and then provides training features on the model. The layers consist of 2D Convolution layer (Conv2D), Max Pooling for 2D spatial data layer (MaxPooling2D), Dropout layer (Dropout), Flatten layer (Flatten) and Dense layer (Dense). Conv2D is used to create a convolution kernel that is convolved with the layer input to produce a tensor of outputs. The filter will slide over the image input and perform an element-wise multiplication. It will then sum all the results into a single output pixel. Furthermore, MaxPooling2D layer is used to down-sample the images, reducing its dimensionality. As a result, it will help reduce the possibility of overfitting and decrease the computational cost by reducing the number of trainable parameters. Dropout layer is also used to prevent overfitting by forcing the layers to take more or less responsibility by taking a probabilistic approach instead of learning the statistical noise. Flatten layer is used to convert the result of 2D arrays from MaxPooling2D layers into a long continuous linear vector which will then be passed into the fully connected layer and with the use of Dense layer, it will be added to the neural network.

```
from tensorflow.keras import layers

model = tf.keras.Sequential([

   layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),

  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Conv2D(128, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Dropout(0.5),
  layers.Flatten(),

  layers.Dense(128, activation='relu'),
  layers.Dense(4,activation="softmax")
])
```

Figure 4. *Proposed Model Structure*

In the figure above, we can see all the layers that are added into the sequential model. Firstly, we will need to re-scale the image pixels by converting the pixels ranging from 0 to 255 into 0

to 1 by dividing it with 1/255, also known as normalisation. This will allow the image to contribute more evenly to the total loss. In the first layer (Rescaling), it will also take in the input shape, which has been pre-defined as 256 by 256 by 3. Then, the Conv2D and MaxPooling2D layers will be applied. As seen from the figure above, the filter size increases as the layers go deeper. This is because at each layer, there are several patterns to be captured such as edges, corners, dots and many more. The deeper the layer goes, it will combine all the patterns and since it becomes more and more complex, there will be larger combinations of patterns to be captured. Hence, the filter size should increase. The padding "same" is applied such that the input image gets fully covered by the filter and the activation "relu" is used as it will convert negative input value to 0, disabling the activation of all neurons at the same time and therefore preventing exponential growth in the computation to operate the neural network. In the last Dense layer, the number 4 indicated the number of classes, and softmax is used for multi-class classification problems.

```
model.compile(optimizer="Adam",
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=["accuracy"])
model.summary()
```

Figure 5. *Loss and metrics for proposed model*

The model will then be compiled by using Adam optimizer since it requires less parameters to be fine tuned and saves the computation time. Since our model consists of 4 classes, we will use SparseCategoricalCrossentropy to compute the loss and the metrics that we will use for analysis is the accuracy. The number of iterations, or epochs, will be set at 10 initially to test for the loss and accuracy of both training and testing data.

4. Model Evaluation and Deployment using AWS-SageMaker

AWS-SageMaker is more beneficial than running the model on a local machine as it contains supplementary software that aids machine learning and decreases the training time significantly. The AWS-SageMaker used in this model has the version of 2.127.0.

```
ValueError: Exception encountered when calling layer "max_pooling2d_1" (type MaxPooling2D).

Negative dimension size caused by subtracting 2 from 1 for '{{node max_pooling2d_1/MaxPool}} = MaxPool[T=DT_FLOAT,
data_format="NCHW", explicit_paddings=[], ksize=[1, 1, 2, 2], padding="VALID", strides=[1, 1, 2, 2]](Placeholder)'
with input shapes: [?,32,128,1].

Call arguments received by layer "max_pooling2d_1" (type MaxPooling2D):
  • inputs=tf.Tensor(shape=(None, 32, 128, 1), dtype=float32)
```

Figure 6. *Error Code on AWS SageMaker*

However, by running the model on AWS Sage-Maker, it shows that there has been an error while building the model in the MaxPooling2D layer due to the negative dimension size.

However, by running the same model on a local machine, no error is found and the model is able to compute the loss and accuracy of the training and testing data. Several data preprocessing have also been implemented, such as by changing the splitting technique and ratio but none of them aid the model to run on SageMaker. Therefore, the model will be run on a local machine for this project. The figure below displays the loss and accuracy of the model.

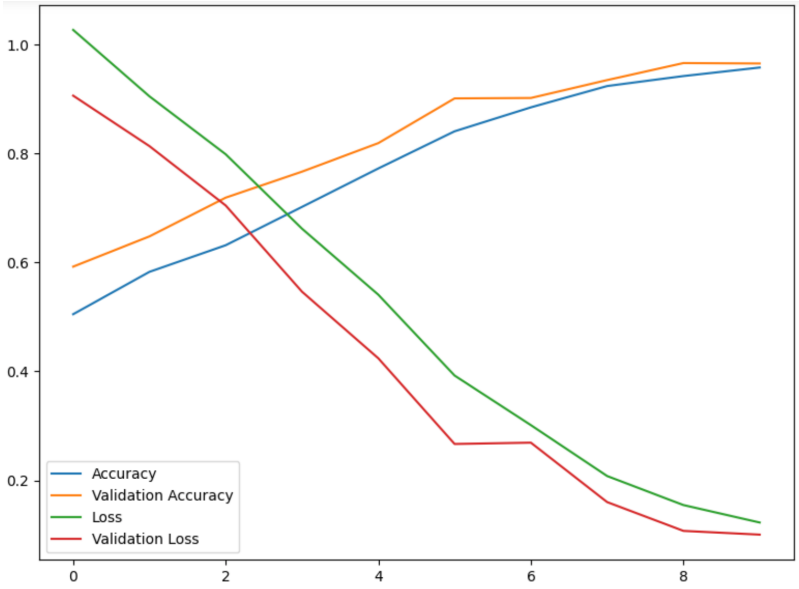


Figure 7. *Model's performance*

The training data performs relatively well, with an accuracy of approximately 95.7% and loss of 0.123 at 10 iterations. The loss value indicates the model's prediction on a single example. Therefore, the lower the value, the better the model performs, as also seen from the relatively high accuracy. The model can be improved further by running more iterations, but since each iteration takes approximately 4 minutes, it will be inefficient to increase the performance marginally at the expense of significant increase in the training time. Furthermore, the val_loss and val_accuracy, which indicates the model's prediction on the testing data, also proves that the model is rather accurate, with the loss at 0.100 and accuracy of 96.5%. This means that the model is considerably reliable and performs even better on the testing data. Overall, we can conclude that the model is able to provide us insights about the possibility of someone having been diagnosed with AD. This model can be used in the preliminary stage of detection of AD, which will allow the potential patient to take precautionary measures to reduce the degenerative process. However, in terms of implementing the model in real-life applications to detect AD, such as hospitals or clinics, it might be a little risky, considering that there may be other external factors that may affect the MRI images such as the patient's medical history. Studies and experts have also questioned the accuracy of MRI, as one study by Chen et. al. (2005) states

that MRI has an accuracy of less than 70%. Therefore, with an input of low accuracy, the model might also predict the result inaccurately. Therefore, on one hand, the model has met the research objective of detecting potential AD in patients early, but the reliability can still be improved by conducting deeper research on improving MRI and the images it produces. This will then enable the model to learn and predict the result at higher accuracy and therefore be able to be implemented in the real world.

5. AWS-SageMaker Discussion

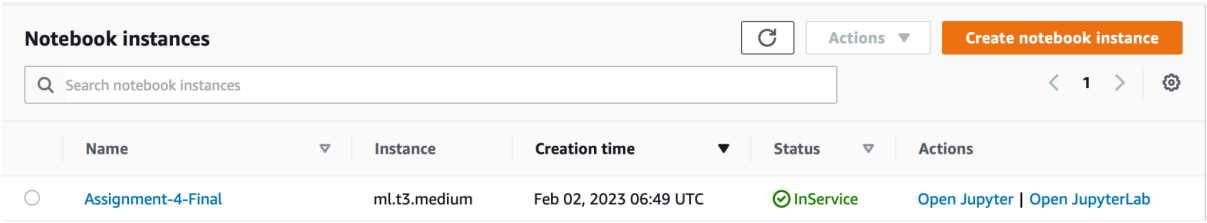


Figure 8. *AWS SageMaker Notebook Instance*

The notebook instance type that is used in the AWS-SageMaker is ml.t3.medium. T3 is used since it is "aimed at general purpose workloads, but at lower price point" (Cahill, 2018). The size chosen is medium since the data that is being dealt with is medium in size. In addition, it is the default instance type for CPU-based SageMaker images, and is also free of charge for the first 250 hours. Since the model is run at a local machine without the aid of SageMaker, we are unable to find the cost and computation time as seen on the example below. Furthermore, in this case, no endpoint is visible on the AWS SageMaker.

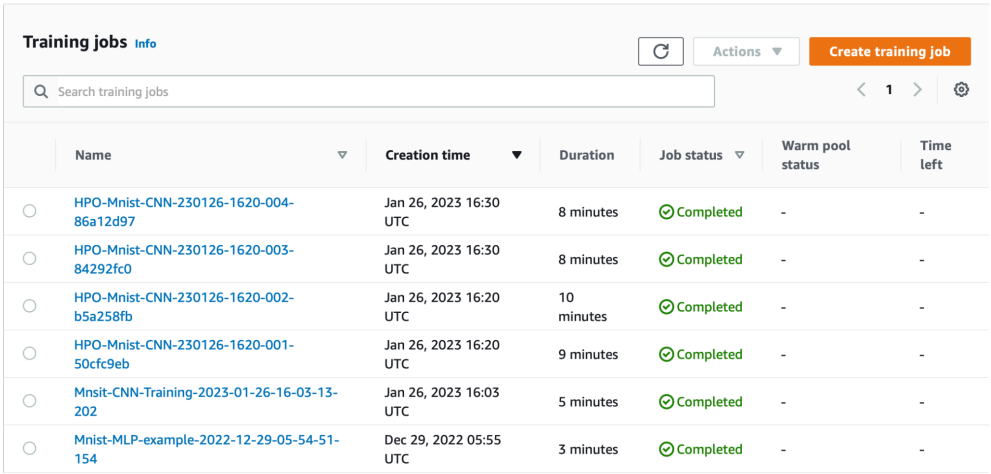


Figure 9. *Example of cost and computation time on AWS SageMaker*

However, by using the datetime module, we can find out the time it takes to complete the job to fit the model. As seen below, the model requires approximately 40 minutes to run 10 epochs, with each epoch taking 240 seconds to complete.

```
epochs = 10
start_time = datetime.now()
history = model.fit(train_data,
                    epochs=epochs,
                    validation_data=val_data,
                    batch_size=batch_size)
end_time = datetime.now()
```

```
print('Duration for personal model: {}'.format(end_time - start_time))
```

```
Duration for personal model: 0:39:51.679972
```

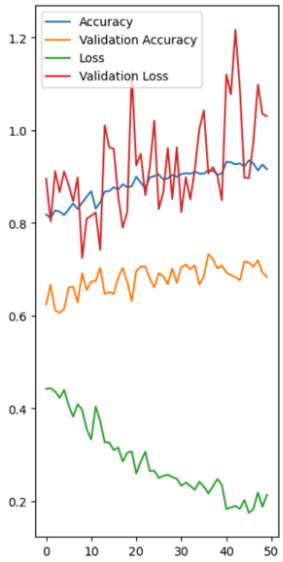
Figure 10. *Cost and computation time of proposed model*

It can also be noted that should the model work on AWS SageMaker, the total computation time could have been decreased significantly as it runs on the cloud as opposed to local host machine, in which the GPU, CPU and RAM might be more capable to run the model at higher efficiency rate.

6. Transfer-learning and Model Comparison

Transfer learning is implemented as it can achieve optimal performance more efficiently than building a new model as it uses a pre-trained model from Keras. It then can be used for fine tuning of the parameters to improve the performance of our own personal model. In this project, there are 3 types of pre-trained models that are used - ResNet50V2, BiT and EfficientNetB0. In each case, the weights are frozen so as to prevent overfitting.

6.1. ResNet50V2




```
Epoch 46/50
67/67 [==============================] - 461s 7s/step - loss: 0.1749
- accuracy: 0.9356 - val_loss: 0.8974 - val_accuracy: 0.7145
Epoch 47/50
67/67 [==============================] - 456s 7s/step - loss: 0.1829
- accuracy: 0.9285 - val_loss: 0.9759 - val_accuracy: 0.7060
Epoch 48/50
67/67 [==============================] - 444s 7s/step - loss: 0.2186
- accuracy: 0.9135 - val_loss: 1.0988 - val_accuracy: 0.7201
Epoch 49/50
67/67 [==============================] - 444s 7s/step - loss: 0.1880
- accuracy: 0.9262 - val_loss: 1.0354 - val_accuracy: 0.6934
Epoch 50/50
67/67 [==============================] - 449s 7s/step - loss: 0.2136
- accuracy: 0.9161 - val_loss: 1.0307 - val_accuracy: 0.6835
```

Figure 11. *Performance of ResNet50V2*

Out of the three models, ResNet50V2 produces the result with the highest accuracy and minimum loss, both on the training and testing data. The accuracy of ResNet50V2 floats around 92%, with a maximum value recorded at 93.6% and the loss is 0.175 at the 46th iteration. However, the testing set performs rather mediocre on the model as the accuracy is only recorded at approximately 71.5% despite the loss being relatively low. This is considerably similar to the benchmark model as seen in Section 4 in which the accuracy and loss stand at 95.7% and 0.123 respectively. Hence, some fine tuning in the model that is built upon ResNet50V2 could be done to increase the reliability of the model in predicting the output. However, it should also be noted that the time it takes to run the ResNet50V2 model is significantly higher than the benchmark model. Therefore, it may not be the best decision should time be a concerning factor. As seen from the time taken to run the ResNet50V2, it takes approximately 450 seconds, and it requires more iterations to reach the desired level of accuracy.

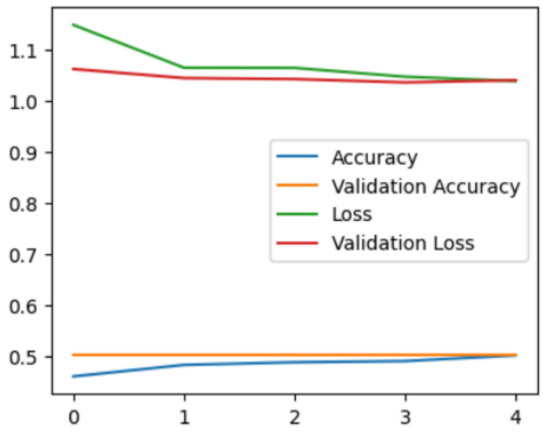
6.2. BiT

```
Epoch 1/4
134/134 [==============================] - 993s 7s/step - loss: 1.001
7 - accuracy: 0.5245 - val_loss: 0.9224 - val_accuracy: 0.5668
Epoch 2/4
134/134 [==============================] - 936s 7s/step - loss: 0.917
9 - accuracy: 0.5664 - val_loss: 0.8909 - val_accuracy: 0.5696
Epoch 3/4
134/134 [==============================] - 923s 7s/step - loss: 0.873
1 - accuracy: 0.5814 - val_loss: 0.8920 - val_accuracy: 0.5823
Epoch 4/4
134/134 [==============================] - 949s 7s/step - loss: 0.869
3 - accuracy: 0.5857 - val_loss: 0.8917 - val_accuracy: 0.5809
```

Figure 12. *Performance of BiT*

The second model, BiT, short for Big Transfer, is used to compare the performance to ResNet50V2. However, as seen from the result, it performs worse in comparison to ResNet50V2 and the benchmark model at accuracy of only approximately 55% and loss 0.9 as compared to Resnet50V2 and the time taken is twice as long as ResNet50V2 and five times as compared to the benchmark model.

6.3. EfficientNetB0




```
Epoch 1/5
134/134 [==============================] - 258s 2s/step - loss: 1.147
4 - accuracy: 0.4596 - val_loss: 1.0611 - val_accuracy: 0.5007
Epoch 2/5
134/134 [==============================] - 234s 2s/step - loss: 1.063
5 - accuracy: 0.4821 - val_loss: 1.0434 - val_accuracy: 0.5007
Epoch 3/5
134/134 [==============================] - 266s 2s/step - loss: 1.063
3 - accuracy: 0.4872 - val_loss: 1.0414 - val_accuracy: 0.5007
Epoch 4/5
134/134 [==============================] - 264s 2s/step - loss: 1.046
0 - accuracy: 0.4896 - val_loss: 1.0349 - val_accuracy: 0.5007
Epoch 5/5
134/134 [==============================] - 245s 2s/step - loss: 1.037
6 - accuracy: 0.5008 - val_loss: 1.0391 - val_accuracy: 0.5007
```

Figure 13. *Performance of EfficientNetB0*

Lastly, EfficientNetB0 is implemented to be compared against ResNet50V2, BiT and benchmark model. Overall, as seen from Figure 13, it can be concluded that EfficientNetB0 does not produce the desired level of accuracy and loss, and hence despite being the most efficient model which has a running time equal to the benchmark model at 250 seconds per epoch, it is not able to improve the accuracy and loss value as significantly. Therefore, this model does not prove to be helpful when conducting the fine tuning of the parameters.

In conclusion, the harmonization of benchmark model and ResNet50V2 together will allow us to increase the reliability of the model as they are already able to classify the images feed into them correctly in majority of the time. Furthermore, the use of ResnET50v2 to conduct transfer learning will significantly increase the ease in which the model can be constructed as it can serve as the base layer, hence reducing the time required to build the model from scratch. It is also used to improve the performance of the current model, by altering the hyperparameters used and therefore prove to be exceptionally useful in the future work to detect not only AD but also various other diseases such as cancer or other illness related to organs, tissues or skeletal system.

7. References

Barber, R. C. (2010). Biomarkers for early detection of Alzheimer disease. *Journal of Osteopathic Medicine*, *110*(s98), 10-15.

Chen, C. C., Lee, R. C., Lin, J. K., Wang, L. W., & Yang, S. H. (2005). How accurate is magnetic resonance imaging in restaging rectal cancer in patients receiving preoperative combined chemoradiotherapy?. *Diseases of the colon & rectum*, *48*, 722-728.

Grabher, B. J. (2018). Effects of Alzheimer disease on patients and their family. *Journal of nuclear medicine technology*, *46*(4), 335-340.

Guerchet, M., Prince, M., & Prina, M. (2020). Numbers of people with dementia worldwide: An update to the estimates in the World Alzheimer Report 2015.

Leifer, B. P. (2003). Early diagnosis of Alzheimer's disease: clinical and economic benefits. *Journal of the American Geriatrics Society*, *51*(5s2), S281-S288.

Salehi, A. W., Baglat, P., Sharma, B. B., Gupta, G., & Upadhya, A. (2020, September). A CNN model: earlier diagnosis and classification of Alzheimer disease using MRI. In *2020 International Conference on Smart Electronics and Communication (ICOSEC)* (pp. 156-161). IEEE.

8. Appendix

A4_Final

February 3, 2023

1 Data Preprocessing

[282]:
```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import os
import pathlib
import random
```

[283]:
```
path = '/Users/richardreynard/Downloads/AY 2022/SP53:22/MA3832/Assignment4/
 ↪Dataset/'
data_dir = pathlib.Path(path)
```

[284]:
```
class_names = np.array([sorted(item.name for item in data_dir.glob("*"))])
class_names
```

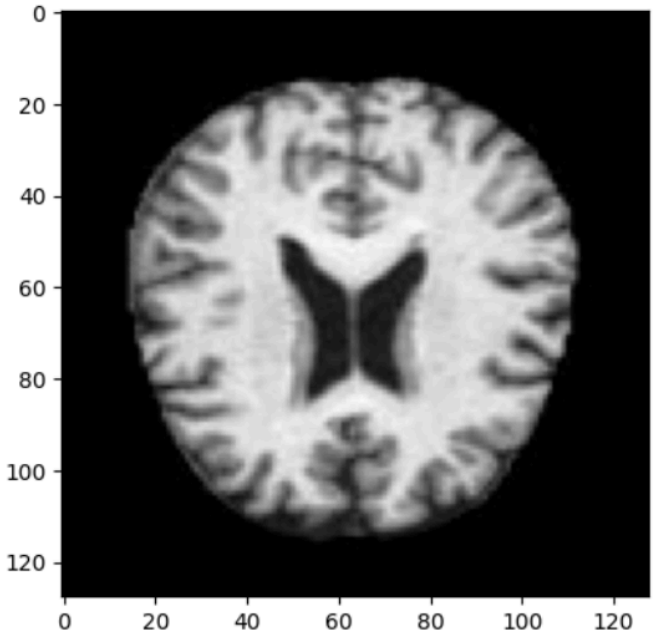
[284]:
```
array([['.DS_Store', 'Mild_Demented', 'Moderate_Demented',
        'Non_Demented', 'Very_Mild_Demented']], dtype='<U18')
```

[285]:
```
imageCount = len(list(data_dir.glob("*/*.jpg") ))
imageCount
```

[285]:
```
6400
```

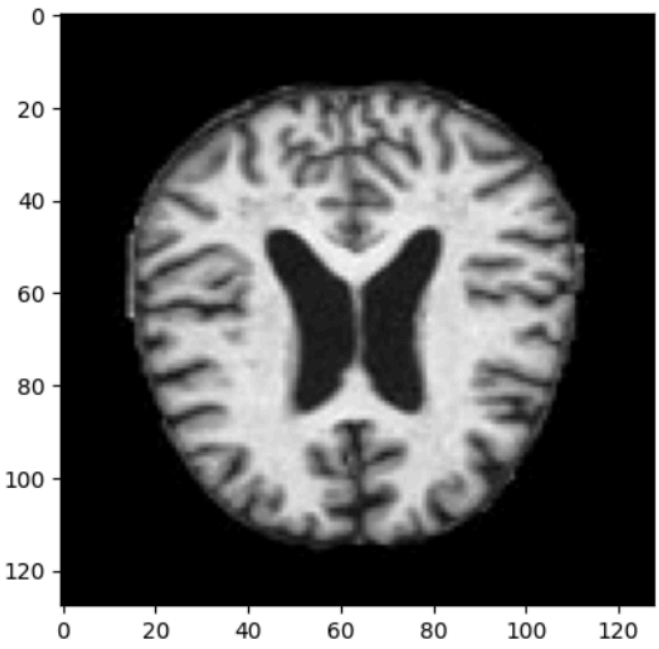
[286]:
```
import cv2
from matplotlib import pyplot as plt
non_demented_image = cv2.imread("Dataset/Non_Demented/non.jpg")
plt.imshow(non_demented_image)
```

[286]:
```
<matplotlib.image.AxesImage at 0x7fd257fb1c40>
```



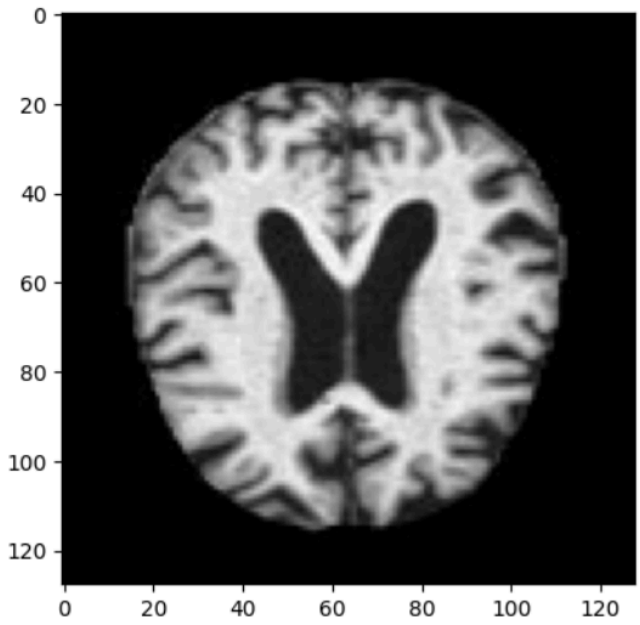
[287]:
```
very_mild_demented_image = cv2.imread("Dataset/Very_Mild_Demented/verymild.jpg")
plt.imshow(very_mild_demented_image)
```

[287]: <matplotlib.image.AxesImage at 0x7fd1da92fc10>



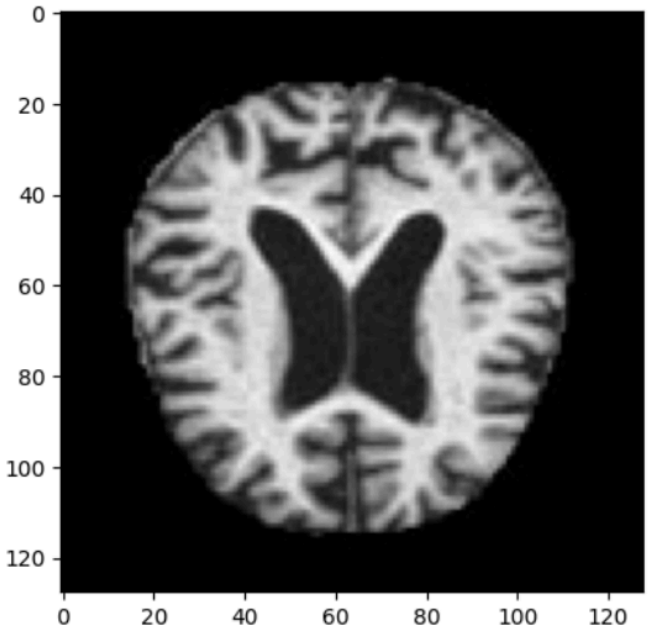
[288]:
```
mild_demented_image = cv2.imread("Dataset/Mild_Demented/mild.jpg")
plt.imshow(mild_demented_image)
```

[288]:
```
<matplotlib.image.AxesImage at 0x7fd2b9eb8700>
```



[289]:
```
moderate_demented_image = cv2.imread("Dataset/Moderate_Demented/moderate.jpg")
plt.imshow(moderate_demented_image)
```

[289]: <matplotlib.image.AxesImage at 0x7fce41252700>



[296]:
```
batch_size = 32
img_height = 256
img_width = 256
```

[297]:
```
from tensorflow.keras.utils import image_dataset_from_directory
from tensorflow.keras.utils import image_dataset_from_directory

train_data = image_dataset_from_directory(
                  data_dir,
                  validation_split=0.2,
                  subset="training",
                  seed=123,
                  image_size=(img_height, img_width),
                  batch_size=batch_size)


val_data = image_dataset_from_directory(data_dir,
                                        validation_split=0.2,
                                        subset="validation",
                                        seed=123,
                                        image_size=(img_height,img_width),
                                        batch_size=batch_size)
```

```
Found 6400 files belonging to 4 classes.
Using 5120 files for training.
Found 6400 files belonging to 4 classes.
Using 1280 files for validation.
```

2 Build Model

[302]:
```
from tensorflow.keras import layers

model = tf.keras.Sequential([

   layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),

  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Conv2D(128, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),

  layers.Dropout(0.5),
  layers.Flatten(),

  layers.Dense(128, activation='relu'),
  layers.Dense(4,activation="softmax")
])
```

[303]:
```
model.compile(optimizer="Adam",
            loss=tf.keras.losses.SparseCategoricalCrossentropy(),
            metrics=["accuracy"])
model.summary()
```

```
Model: "sequential_61"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_32 (Rescaling)    (None, 256, 256, 3)       0

 conv2d_106 (Conv2D)         (None, 256, 256, 16)      448

 max_pooling2d_121 (MaxPooli  (None, 128, 128, 16)     0
 ng2D)
```

```
conv2d_107 (Conv2D)          (None, 128, 128, 32)      4640

max_pooling2d_122 (MaxPooli  (None, 64, 64, 32)       0
ng2D)

conv2d_108 (Conv2D)          (None, 64, 64, 64)        18496

max_pooling2d_123 (MaxPooli  (None, 32, 32, 64)       0
ng2D)

conv2d_109 (Conv2D)          (None, 32, 32, 128)       73856

max_pooling2d_124 (MaxPooli  (None, 16, 16, 128)      0
ng2D)

dropout_44 (Dropout)         (None, 16, 16, 128)       0

flatten_44 (Flatten)         (None, 32768)             0

dense_125 (Dense)            (None, 128)               4194432

dense_126 (Dense)            (None, 4)                 516

=================================================================
Total params: 4,292,388
Trainable params: 4,292,388
Non-trainable params: 0
_________________________________________________________________
```

[304]:
```
epochs = 10
history = model.fit(train_data,
                    epochs=epochs,
                    validation_data=val_data,
                    batch_size=batch_size)
```

```
Epoch 1/10
160/160 [==============================] - 220s 1s/step - loss: 1.0449 -
accuracy: 0.4977 - val_loss: 0.9187 - val_accuracy: 0.5703
Epoch 2/10
160/160 [==============================] - 206s 1s/step - loss: 0.9055 -
accuracy: 0.5680 - val_loss: 0.8020 - val_accuracy: 0.6367
Epoch 3/10
160/160 [==============================] - 222s 1s/step - loss: 0.7925 -
accuracy: 0.6359 - val_loss: 0.7002 - val_accuracy: 0.7133
Epoch 4/10
160/160 [==============================] - 213s 1s/step - loss: 0.6673 -
accuracy: 0.7014 - val_loss: 0.5533 - val_accuracy: 0.7719
```

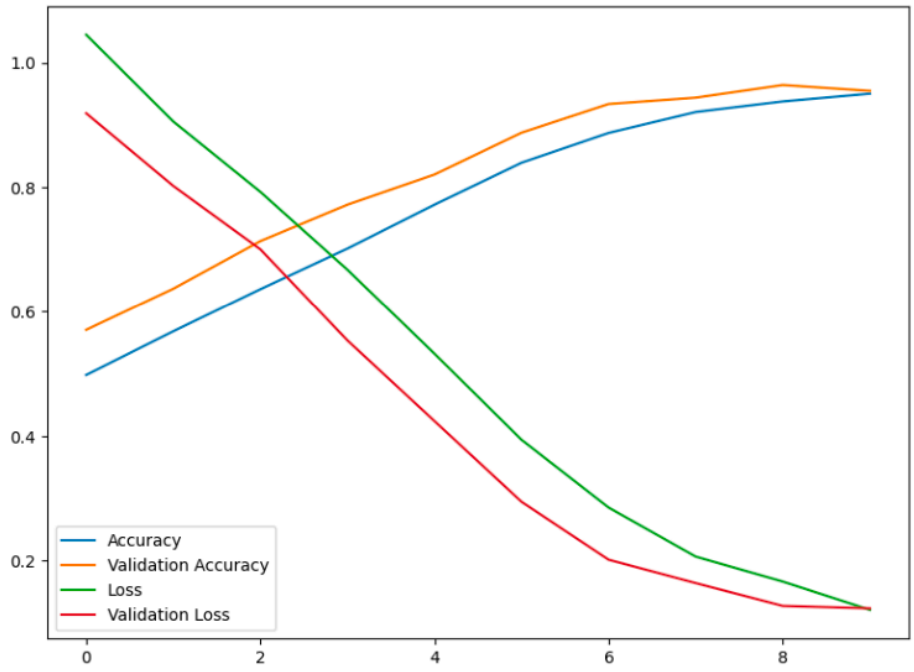
```
Epoch 5/10
160/160 [==============================] - 204s 1s/step - loss: 0.5319 -
accuracy: 0.7719 - val_loss: 0.4236 - val_accuracy: 0.8203
Epoch 6/10
160/160 [==============================] - 200s 1s/step - loss: 0.3935 -
accuracy: 0.8393 - val_loss: 0.2939 - val_accuracy: 0.8875
Epoch 7/10
160/160 [==============================] - 231s 1s/step - loss: 0.2847 -
accuracy: 0.8871 - val_loss: 0.2009 - val_accuracy: 0.9336
Epoch 8/10
160/160 [==============================] - 232s 1s/step - loss: 0.2060 -
accuracy: 0.9207 - val_loss: 0.1634 - val_accuracy: 0.9438
Epoch 9/10
160/160 [==============================] - 234s 1s/step - loss: 0.1659 -
accuracy: 0.9377 - val_loss: 0.1267 - val_accuracy: 0.9641
Epoch 10/10
160/160 [==============================] - 231s 1s/step - loss: 0.1204 -
accuracy: 0.9502 - val_loss: 0.1228 - val_accuracy: 0.9547
```

[312]:
```
acc = history.history['accuracy']
val_acc =  history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(32,32))
plt.subplot(4,3,4)
plt.plot(epochs_range,acc,label='Accuracy')
plt.plot(epochs_range,val_acc,label="Validation Accuracy")
plt.plot(epochs_range,loss,label='Loss')
plt.plot(epochs_range,val_loss,label="Validation Loss")
plt.legend()
```

[312]:
```
<matplotlib.legend.Legend at 0x7fcf999b5520>
```



3 Transfer Learning

[313]:
```
import tensorflow as tf
from keras_preprocessing.image import ImageDataGenerator
import os
import cv2
import random
from matplotlib import pyplot as plt
import pathlib
import pandas as pd
import numpy as np
```

[239]:
```
# train_dir = '/Users/richardreynard/Downloads/Dataset3/Train_Set'
# test_dir = '/Users/richardreynard/Downloads/Dataset3/Test_Set'
```

[314]:
```
train_dir = '/Users/richardreynard/Downloads/Dataset_BiT/Training Data'
test_dir = '/Users/richardreynard/Downloads/Dataset_BiT/Testing Data'
val_dir = '/Users/richardreynard/Downloads/Dataset_BiT/Validation Data'
```

[238]:
```
# from keras.preprocessing.image import ImageDataGenerator as IDG

# # Initialize image data generator
# train_gen = IDG(rescale=1/255, rotation_range=10, horizontal_flip=True,␣
 ↪vertical_flip=False)
# test_gen = IDG(rescale=1/255, rotation_range=10, horizontal_flip=True,␣
 ↪vertical_flip=False)
# valid_gen = IDG(rescale=1/255)

# # Load the datasets
# train_ds = train_gen.flow_from_directory(train_dir, shuffle=True,␣
 ↪batch_size=64, target_size=(256,256), class_mode='binary')
# test_ds = test_gen.flow_from_directory(train_dir, shuffle=True,␣
 ↪batch_size=64, target_size=(256,256), class_mode='binary')
# valid_ds = test_gen.flow_from_directory(test_dir, shuffle=True,␣
 ↪batch_size=32, target_size=(256,256), class_mode='binary')
```

```
Found 5120 images belonging to 4 classes.
Found 5120 images belonging to 4 classes.
Found 1280 images belonging to 4 classes.
```

[240]:
```
from keras.preprocessing.image import ImageDataGenerator as IDG

# Initialize image data generator
train_gen = IDG(rescale=1/255, rotation_range=10, horizontal_flip=True,␣
 ↪vertical_flip=False)
test_gen = IDG(rescale=1/255, rotation_range=10, horizontal_flip=True,␣
 ↪vertical_flip=False)
valid_gen = IDG(rescale=1/255)

# Load the datasets
train_ds = train_gen.flow_from_directory(train_dir, shuffle=True,␣
 ↪batch_size=64, target_size=(256,256), class_mode='binary')
test_ds = test_gen.flow_from_directory(test_dir, shuffle=True, batch_size=64,␣
 ↪target_size=(256,256), class_mode='binary')
valid_ds = test_gen.flow_from_directory(val_dir, shuffle=True, batch_size=32,␣
 ↪target_size=(256,256), class_mode='binary')
```

```
Found 4267 images belonging to 4 classes.
Found 1422 images belonging to 4 classes.
Found 711 images belonging to 4 classes.
```

[241]:
```
from keras.layers import Dense, GlobalAveragePooling2D as GAP, Dropout
from keras.models import load_model, Sequential

# Pre Trained Models
```

```
from tensorflow.keras.applications import ResNet50V2, InceptionV3, Xception,␣
↪ResNet50, ResNet152V2
```

3.1 Using ResNet50V2 - Most Accurate

[242]:
```
# Base Model
base = ResNet50V2(include_top=False, input_shape=(256,256,3))
base.trainable = False

# Model Architecture
model = tf.keras.Sequential([
    base,
    GAP(),
    Dense(1024, kernel_initializer='he_normal', activation='relu'),
    Dense(512, kernel_initializer='he_normal', activation='relu'),
    Dropout(0.4),
    Dense(4,activation="softmax"),
])


# Compile
model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=tf.keras.optimizers.Adam(learning_rate=2e-3),
    metrics=['accuracy']
)
```

[243]:
```
epochs = 20
history = model.fit(
            train_ds,
            validation_data = valid_ds,
            epochs = epochs)
```

```
Epoch 1/20
67/67 [==============================] - 399s 6s/step - loss: 1.5325 - accuracy:
0.4870 - val_loss: 0.9243 - val_accuracy: 0.5682
Epoch 2/20
67/67 [==============================] - 413s 6s/step - loss: 0.9041 - accuracy:
0.5742 - val_loss: 0.8858 - val_accuracy: 0.5851
Epoch 3/20
67/67 [==============================] - 426s 6s/step - loss: 0.8566 - accuracy:
0.5948 - val_loss: 0.8882 - val_accuracy: 0.5767
Epoch 4/20
67/67 [==============================] - 492s 7s/step - loss: 0.8359 - accuracy:
0.6072 - val_loss: 0.8668 - val_accuracy: 0.5977
Epoch 5/20
67/67 [==============================] - 620s 9s/step - loss: 0.8000 - accuracy:
```

```
0.6307 - val_loss: 0.8594 - val_accuracy: 0.5668
Epoch 6/20
67/67 [==============================] - 469s 7s/step - loss: 0.7669 - accuracy:
0.6445 - val_loss: 0.8376 - val_accuracy: 0.6020
Epoch 7/20
67/67 [==============================] - 456s 7s/step - loss: 0.7536 - accuracy:
0.6478 - val_loss: 0.8160 - val_accuracy: 0.6188
Epoch 8/20
67/67 [==============================] - 491s 7s/step - loss: 0.7149 - accuracy:
0.6773 - val_loss: 0.8063 - val_accuracy: 0.6357
Epoch 9/20
67/67 [==============================] - 508s 8s/step - loss: 0.6793 - accuracy:
0.7003 - val_loss: 0.8001 - val_accuracy: 0.6414
Epoch 10/20
67/67 [==============================] - 497s 7s/step - loss: 0.6428 - accuracy:
0.7209 - val_loss: 0.8043 - val_accuracy: 0.6371
Epoch 11/20
67/67 [==============================] - 501s 7s/step - loss: 0.6205 - accuracy:
0.7270 - val_loss: 0.7921 - val_accuracy: 0.6385
Epoch 12/20
67/67 [==============================] - 513s 8s/step - loss: 0.6232 - accuracy:
0.7288 - val_loss: 0.7934 - val_accuracy: 0.6371
Epoch 13/20
67/67 [==============================] - 503s 8s/step - loss: 0.6022 - accuracy:
0.7413 - val_loss: 0.8621 - val_accuracy: 0.6343
Epoch 14/20
67/67 [==============================] - 513s 8s/step - loss: 0.5771 - accuracy:
0.7570 - val_loss: 0.8078 - val_accuracy: 0.6442
Epoch 15/20
67/67 [==============================] - 490s 7s/step - loss: 0.5540 - accuracy:
0.7628 - val_loss: 0.8008 - val_accuracy: 0.6582
Epoch 16/20
67/67 [==============================] - 437s 7s/step - loss: 0.5424 - accuracy:
0.7654 - val_loss: 0.7863 - val_accuracy: 0.6428
Epoch 17/20
67/67 [==============================] - 481s 7s/step - loss: 0.5419 - accuracy:
0.7628 - val_loss: 0.8347 - val_accuracy: 0.6245
Epoch 18/20
67/67 [==============================] - 487s 7s/step - loss: 0.4900 - accuracy:
0.7968 - val_loss: 0.8192 - val_accuracy: 0.6498
Epoch 19/20
67/67 [==============================] - 496s 7s/step - loss: 0.4868 - accuracy:
0.7935 - val_loss: 0.8406 - val_accuracy: 0.6568
Epoch 20/20
67/67 [==============================] - 478s 7s/step - loss: 0.4421 - accuracy:
0.8188 - val_loss: 0.7834 - val_accuracy: 0.6793
```

[244]:
```
epochs = 50
history = model.fit(
            train_ds,
            validation_data = valid_ds,
            epochs = epochs)
```

```
Epoch 1/50
67/67 [==============================] - 437s 7s/step - loss: 0.4423 - accuracy:
0.8186 - val_loss: 0.8957 - val_accuracy: 0.6245
Epoch 2/50
67/67 [==============================] - 464s 7s/step - loss: 0.4435 - accuracy:
0.8106 - val_loss: 0.8037 - val_accuracy: 0.6667
Epoch 3/50
67/67 [==============================] - 464s 7s/step - loss: 0.4367 - accuracy:
0.8275 - val_loss: 0.9119 - val_accuracy: 0.6118
Epoch 4/50
67/67 [==============================] - 445s 7s/step - loss: 0.4227 - accuracy:
0.8245 - val_loss: 0.8668 - val_accuracy: 0.6062
Epoch 5/50
67/67 [==============================] - 442s 7s/step - loss: 0.4400 - accuracy:
0.8172 - val_loss: 0.9114 - val_accuracy: 0.6146
Epoch 6/50
67/67 [==============================] - 445s 7s/step - loss: 0.4066 - accuracy:
0.8294 - val_loss: 0.8820 - val_accuracy: 0.6610
Epoch 7/50
67/67 [==============================] - 455s 7s/step - loss: 0.3823 - accuracy:
0.8423 - val_loss: 0.8471 - val_accuracy: 0.6624
Epoch 8/50
67/67 [==============================] - 470s 7s/step - loss: 0.4095 - accuracy:
0.8306 - val_loss: 0.8988 - val_accuracy: 0.6287
Epoch 9/50
67/67 [==============================] - 478s 7s/step - loss: 0.3966 - accuracy:
0.8434 - val_loss: 0.7246 - val_accuracy: 0.6920
Epoch 10/50
67/67 [==============================] - 481s 7s/step - loss: 0.3556 - accuracy:
0.8570 - val_loss: 0.8094 - val_accuracy: 0.6554
Epoch 11/50
67/67 [==============================] - 482s 7s/step - loss: 0.3332 - accuracy:
0.8688 - val_loss: 0.8156 - val_accuracy: 0.6737
Epoch 12/50
67/67 [==============================] - 498s 7s/step - loss: 0.4041 - accuracy:
0.8310 - val_loss: 0.8227 - val_accuracy: 0.6751
Epoch 13/50
67/67 [==============================] - 484s 7s/step - loss: 0.3727 - accuracy:
0.8449 - val_loss: 0.7421 - val_accuracy: 0.7032
Epoch 14/50
67/67 [==============================] - 513s 8s/step - loss: 0.3273 - accuracy:
```

```
0.8683 - val_loss: 1.0109 - val_accuracy: 0.6470
Epoch 15/50
67/67 [==============================] - 546s 8s/step - loss: 0.3265 - accuracy:
0.8688 - val_loss: 0.9632 - val_accuracy: 0.6512
Epoch 16/50
67/67 [==============================] - 484s 7s/step - loss: 0.3104 - accuracy:
0.8774 - val_loss: 0.9600 - val_accuracy: 0.6470
Epoch 17/50
67/67 [==============================] - 469s 7s/step - loss: 0.3155 - accuracy:
0.8723 - val_loss: 0.8573 - val_accuracy: 0.6821
Epoch 18/50
67/67 [==============================] - 464s 7s/step - loss: 0.2857 - accuracy:
0.8835 - val_loss: 0.7902 - val_accuracy: 0.7032
Epoch 19/50
67/67 [==============================] - 477s 7s/step - loss: 0.3040 - accuracy:
0.8777 - val_loss: 0.8240 - val_accuracy: 0.6737
Epoch 20/50
67/67 [==============================] - 464s 7s/step - loss: 0.3072 - accuracy:
0.8795 - val_loss: 1.1248 - val_accuracy: 0.6315
Epoch 21/50
67/67 [==============================] - 453s 7s/step - loss: 0.2597 - accuracy:
0.8999 - val_loss: 0.9245 - val_accuracy: 0.6962
Epoch 22/50
67/67 [==============================] - 449s 7s/step - loss: 0.2850 - accuracy:
0.8875 - val_loss: 0.9493 - val_accuracy: 0.7060
Epoch 23/50
67/67 [==============================] - 433s 6s/step - loss: 0.3066 - accuracy:
0.8749 - val_loss: 0.8603 - val_accuracy: 0.7060
Epoch 24/50
67/67 [==============================] - 437s 7s/step - loss: 0.2648 - accuracy:
0.8983 - val_loss: 0.9196 - val_accuracy: 0.6807
Epoch 25/50
67/67 [==============================] - 445s 7s/step - loss: 0.2656 - accuracy:
0.9013 - val_loss: 1.0210 - val_accuracy: 0.6610
Epoch 26/50
67/67 [==============================] - 461s 7s/step - loss: 0.2500 - accuracy:
0.9049 - val_loss: 0.8302 - val_accuracy: 0.6920
Epoch 27/50
67/67 [==============================] - 432s 6s/step - loss: 0.2547 - accuracy:
0.8938 - val_loss: 0.8668 - val_accuracy: 0.6850
Epoch 28/50
67/67 [==============================] - 426s 6s/step - loss: 0.2568 - accuracy:
0.8964 - val_loss: 0.9620 - val_accuracy: 0.6681
Epoch 29/50
67/67 [==============================] - 435s 6s/step - loss: 0.2518 - accuracy:
0.9041 - val_loss: 0.8521 - val_accuracy: 0.7018
Epoch 30/50
67/67 [==============================] - 462s 7s/step - loss: 0.2481 - accuracy:
```

```
0.8995 - val_loss: 0.9633 - val_accuracy: 0.6709
Epoch 31/50
67/67 [==============================] - 458s 7s/step - loss: 0.2333 - accuracy:
0.9058 - val_loss: 0.8234 - val_accuracy: 0.7046
Epoch 32/50
67/67 [==============================] - 461s 7s/step - loss: 0.2401 - accuracy:
0.9077 - val_loss: 0.8987 - val_accuracy: 0.7103
Epoch 33/50
67/67 [==============================] - 465s 7s/step - loss: 0.2328 - accuracy:
0.9063 - val_loss: 0.8510 - val_accuracy: 0.7004
Epoch 34/50
67/67 [==============================] - 451s 7s/step - loss: 0.2246 - accuracy:
0.9105 - val_loss: 0.9158 - val_accuracy: 0.7089
Epoch 35/50
67/67 [==============================] - 446s 7s/step - loss: 0.2418 - accuracy:
0.9070 - val_loss: 1.0032 - val_accuracy: 0.6667
Epoch 36/50
67/67 [==============================] - 451s 7s/step - loss: 0.2316 - accuracy:
0.9067 - val_loss: 1.0431 - val_accuracy: 0.6850
Epoch 37/50
67/67 [==============================] - 447s 7s/step - loss: 0.2166 - accuracy:
0.9138 - val_loss: 0.9070 - val_accuracy: 0.7328
Epoch 38/50
67/67 [==============================] - 452s 7s/step - loss: 0.2319 - accuracy:
0.9131 - val_loss: 0.9205 - val_accuracy: 0.7229
Epoch 39/50
67/67 [==============================] - 457s 7s/step - loss: 0.2477 - accuracy:
0.9041 - val_loss: 0.9027 - val_accuracy: 0.7018
Epoch 40/50
67/67 [==============================] - 447s 7s/step - loss: 0.2351 - accuracy:
0.9070 - val_loss: 0.8493 - val_accuracy: 0.7089
Epoch 41/50
67/67 [==============================] - 443s 7s/step - loss: 0.1828 - accuracy:
0.9320 - val_loss: 1.1205 - val_accuracy: 0.6934
Epoch 42/50
67/67 [==============================] - 453s 7s/step - loss: 0.1863 - accuracy:
0.9313 - val_loss: 1.0778 - val_accuracy: 0.6878
Epoch 43/50
67/67 [==============================] - 462s 7s/step - loss: 0.1897 - accuracy:
0.9266 - val_loss: 1.2172 - val_accuracy: 0.6835
Epoch 44/50
67/67 [==============================] - 444s 7s/step - loss: 0.1833 - accuracy:
0.9288 - val_loss: 1.0887 - val_accuracy: 0.6765
Epoch 45/50
67/67 [==============================] - 446s 7s/step - loss: 0.2024 - accuracy:
0.9222 - val_loss: 0.8976 - val_accuracy: 0.7173
Epoch 46/50
67/67 [==============================] - 461s 7s/step - loss: 0.1749 - accuracy:
```

```
0.9356 - val_loss: 0.8974 - val_accuracy: 0.7145
Epoch 47/50
67/67 [==============================] - 456s 7s/step - loss: 0.1829 - accuracy:
0.9285 - val_loss: 0.9759 - val_accuracy: 0.7060
Epoch 48/50
67/67 [==============================] - 444s 7s/step - loss: 0.2186 - accuracy:
0.9135 - val_loss: 1.0988 - val_accuracy: 0.7201
Epoch 49/50
67/67 [==============================] - 444s 7s/step - loss: 0.1880 - accuracy:
0.9262 - val_loss: 1.0354 - val_accuracy: 0.6934
Epoch 50/50
67/67 [==============================] - 449s 7s/step - loss: 0.2136 - accuracy:
0.9161 - val_loss: 1.0307 - val_accuracy: 0.6835
```

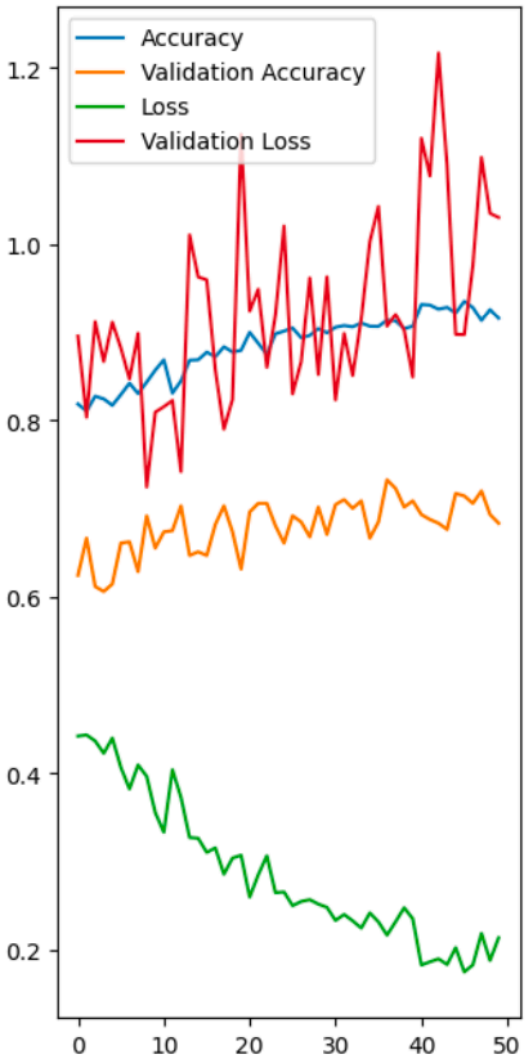
[246]:
```
acc = history.history['accuracy']
val_acc =  history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8,8))
plt.subplot(1,2,2)
plt.plot(epochs_range,acc,label='Accuracy')
plt.plot(epochs_range,val_acc,label="Validation Accuracy")
plt.plot(epochs_range,loss,label='Loss')
plt.plot(epochs_range,val_loss,label="Validation Loss")
plt.legend()
```

[246]:
```
<matplotlib.legend.Legend at 0x7fd27e9af730>
```



3.2 Using BiT - Big Transfer

222]:
```
# import os
# import numpy as np
# import pandas as pd
# from glob import glob
# import tensorflow as tf

# ! pip install "tensorflow>=2.0.0"
```

```
# ! pip install --upgrade tensorflow-hub
# import tensorflow_hub as hub
# from IPython.display import clear_output as cls

# # Data
# from tensorflow.keras.utils import load_img, img_to_array
# from keras.preprocessing.image import ImageDataGenerator

# # Data Visualization
# import plotly.express as px
# import matplotlib.pyplot as plt

# # Model
# from keras.models import Sequential, load_model
# from keras.layers import GlobalAvgPool2D as GAP, Dense, Dropout

# # Callbacks
# from keras.callbacks import EarlyStopping, ModelCheckpoint

# # Pre-Trained Model
# from tensorflow.keras.applications import ResNet50V2
```

[224]:
```
# Check Training Data Information
train_path = '/Users/richardreynard/Downloads/Dataset_BiT/Training Data/'
class_names = sorted(os.listdir(train_path))
n_classes = len(class_names)

# Show
print(f"Total Number of Classes : {n_classes} \nClass Names : {class_names}")
```

```
Total Number of Classes : 5
Class Names : ['.DS_Store', 'Mild_Demented', 'Moderate_Demented',
'Non_Demented', 'Very_Mild_Demented']
```

[225]:
```
# # Check Testing Data Information
test_path = '/Users/richardreynard/Downloads/Dataset_BiT/Testing Data/'
class_names = sorted(os.listdir(test_path))
n_classes = len(class_names)

# Show
print(f"Total Number of Classes : {n_classes} \nClass Names : {class_names}")
```

```
Total Number of Classes : 5
Class Names : ['.DS_Store', 'Mild_Demented', 'Moderate_Demented',
'Non_Demented', 'Very_Mild_Demented']
```

[227]:
```
# Check Validation Data Information
valid_path = '/Users/richardreynard/Downloads/Dataset_BiT/Validation Data/'
class_names = sorted(os.listdir(valid_path))
n_classes = len(class_names)

# Show
print(f"Total Number of Classes : {n_classes} \nClass Names : {class_names}")
```

```
Total Number of Classes : 5
Class Names : ['.DS_Store', 'Mild_Demented', 'Moderate_Demented',
'Non_Demented', 'Very_Mild_Demented']
```

[228]:
```
# Initialize Generator
train_gen = ImageDataGenerator(rescale=1/255., rotation_range=10,␣
 ↪horizontal_flip=True)
valid_gen = ImageDataGenerator(rescale=1/255.)
test_gen = ImageDataGenerator(rescale=1/255)

# Load Data
train_ds = train_gen.flow_from_directory(train_path, class_mode='binary',␣
 ↪target_size=(256,256), shuffle=True, batch_size=32)
valid_ds = valid_gen.flow_from_directory(valid_path, class_mode='binary',␣
 ↪target_size=(256,256), shuffle=True, batch_size=32)
test_ds = test_gen.flow_from_directory(test_path, class_mode='binary',␣
 ↪target_size=(256,256), shuffle=True, batch_size=32)
```

```
Found 4267 images belonging to 4 classes.
Found 711 images belonging to 4 classes.
Found 1422 images belonging to 4 classes.
```

[229]:
```
# Import BiT model
bit_model_url = "https://tfhub.dev/google/bit/m-r50x1/1"
bit_module = hub.KerasLayer(bit_model_url)
```

[230]:
```
model = Sequential([
    bit_module,
    Dense(4, activation='softmax', kernel_initializer='zeros')
], name='bit-custom')
```

[231]:
```
BATCH_SIZE = 32
lr = 1e-3 * BATCH_SIZE/512
print(f"Learning rate : {lr}")
```

```
Learning rate : 6.25e-05
```

[232]:
```
SCHEDULE_BOUNDARIES = [
    200,
```

```
    300,
    400,
]
```

[233]:
```
lr_schedule = tf.keras.optimizers.schedules.PiecewiseConstantDecay(
    boundaries=SCHEDULE_BOUNDARIES,
    values=[
        lr,
        lr * 0.1,
        lr * 0.01,
        lr * 0.001,
    ],
)
optimizer = tf.keras.optimizers.SGD(learning_rate=lr_schedule, momentum=0.9)
```

[234]:
```
model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=optimizer,
    metrics=['accuracy']
)
```

[235]:
```
history = model.fit(train_ds, validation_data=valid_ds, epochs=4)
```

```
Epoch 1/4
134/134 [==============================] - 993s 7s/step - loss: 1.0017 -
accuracy: 0.5245 - val_loss: 0.9224 - val_accuracy: 0.5668
Epoch 2/4
134/134 [==============================] - 936s 7s/step - loss: 0.9179 -
accuracy: 0.5664 - val_loss: 0.8909 - val_accuracy: 0.5696
Epoch 3/4
134/134 [==============================] - 923s 7s/step - loss: 0.8731 -
accuracy: 0.5814 - val_loss: 0.8920 - val_accuracy: 0.5823
Epoch 4/4
134/134 [==============================] - 949s 7s/step - loss: 0.8693 -
accuracy: 0.5857 - val_loss: 0.8917 - val_accuracy: 0.5809
```

3.3 Using EfficientNetB0 - Least Efficient

[315]:
```
train_dir = '/Users/richardreynard/Downloads/Dataset_BiT/Training Data'
test_dir = '/Users/richardreynard/Downloads/Dataset_BiT/Testing Data'
val_dir = '/Users/richardreynard/Downloads/Dataset_BiT/Validation Data'
```

[316]:
```
# Initialize Generator
train_gen = ImageDataGenerator(rescale=1/255., rotation_range=10,␣
 ↪horizontal_flip=True)
valid_gen = ImageDataGenerator(rescale=1/255.)
test_gen = ImageDataGenerator(rescale=1/255)
```

```
# Load Data
train_ds = train_gen.flow_from_directory(train_path, class_mode='binary',␣
 ↪target_size=(256,256), shuffle=True, batch_size=32)
valid_ds = valid_gen.flow_from_directory(valid_path, class_mode='binary',␣
 ↪target_size=(256,256), shuffle=True, batch_size=32)
test_ds = test_gen.flow_from_directory(test_path, class_mode='binary',␣
 ↪target_size=(256,256), shuffle=True, batch_size=32)
```

```
Found 4267 images belonging to 4 classes.
Found 711 images belonging to 4 classes.
Found 1422 images belonging to 4 classes.
```

[317]:
```
base2 = tf.keras.applications.EfficientNetB0(include_top=False,␣
 ↪input_shape=(256,256,3))
base2.trainable = False
model2 = tf.keras.Sequential([
    base2,
    GAP(),
    Dense(1024, kernel_initializer='he_normal', activation='relu'),
    Dropout(0.4),
    Dense(512, kernel_initializer='he_normal', activation='relu'),
    Dropout(0.4),
    Dense(4, activation="softmax")
])


# Compile
model2.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=tf.keras.optimizers.Adam(learning_rate=2e-3),
    metrics=['accuracy']
)
```

[318]:
```
epochs = 5
history = model2.fit(
            train_ds,
            validation_data = valid_ds,
            epochs = epochs)
```

```
Epoch 1/5
134/134 [==============================] - 258s 2s/step - loss: 1.1474 -
accuracy: 0.4596 - val_loss: 1.0611 - val_accuracy: 0.5007
Epoch 2/5
134/134 [==============================] - 234s 2s/step - loss: 1.0635 -
accuracy: 0.4821 - val_loss: 1.0434 - val_accuracy: 0.5007
Epoch 3/5
```

```
134/134 [==============================] - 266s 2s/step - loss: 1.0633 -
accuracy: 0.4872 - val_loss: 1.0414 - val_accuracy: 0.5007
Epoch 4/5
134/134 [==============================] - 264s 2s/step - loss: 1.0460 -
accuracy: 0.4896 - val_loss: 1.0349 - val_accuracy: 0.5007
Epoch 5/5
134/134 [==============================] - 245s 2s/step - loss: 1.0376 -
accuracy: 0.5008 - val_loss: 1.0391 - val_accuracy: 0.5007
```

[326]:
```
acc = history.history['accuracy']
val_acc =  history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(10,8))
plt.subplot(2,2,2)
plt.plot(epochs_range,acc,label='Accuracy')
plt.plot(epochs_range,val_acc,label="Validation Accuracy")
plt.plot(epochs_range,loss,label='Loss')
plt.plot(epochs_range,val_loss,label="Validation Loss")
plt.legend()
```

[326]:
```
<matplotlib.legend.Legend at 0x7fd07539a730>
```