Benjamin Hanim 5/05/2024

Alzheimers Convolutional Neural Network

Step 1: Set Up

To begin my journey into convolutional neural networks, I began by finding an Alzheimers dataset on Kaggle (click here). I originally downloaded the zip file onto my computer and unzipped directly into my files. I started on Jupyter Notebook, but after realizing the processing power necessary for running a multi layered neural network, I switched over to Google Colab by uploading all of the files to Google Drive.

Step 2: Data Analysis

First I needed to determine how many classes the dataset contained, how many images per class in the training and test sets, the shape of the images (to see if they need to be reshaped), view the images themselves to see if I could notice any patterns of symmetry, and lastly see the images' gradients to see if it had to be normalized.

```
os.listdir(data_dir)
```

```
['test', 'train']
```

```
test_path = data_dir+'\\test\\'
train_path = data_dir+'\\train\\'
```

```
os.listdir(test_path)
```

```
['MildDemented', 'ModerateDemented', 'NonDemented', 'VeryMildDemented']
```

```
print("Train set counts:")
print("MildDemented images:", len(os.listdir(train_path + 'MildDemented')))
print("ModerateDemented images:", len(os.listdir(train_path + 'ModerateDemented')))
print("NonDemented images:", len(os.listdir(train_path + 'NonDemented')))
print("VeryMildDemented images:", len(os.listdir(train_path + 'VeryMildDemented')))
print("\nTest set counts:")
print("MildDemented images:", len(os.listdir(test_path + 'MildDemented')))
print("ModerateDemented images:", len(os.listdir(test_path + 'ModerateDemented')))
print("NonDemented images:", len(os.listdir(test_path + 'NonDemented')))
print("VeryMildDemented images:", len(os.listdir(test_path + 'VeryMildDemented')))
```

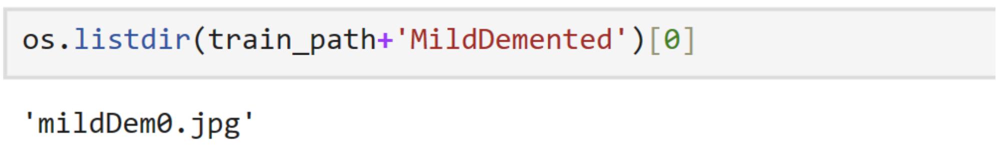
```
Train set counts:
MildDemented images: 717
ModerateDemented images: 52
NonDemented images: 2560
VeryMildDemented images: 1792

Test set counts:
MildDemented images: 179
ModerateDemented images: 12
NonDemented images: 640
VeryMildDemented images: 448
```

```
179/(179+717) #Near 80/20 train/test split
```

```
0.19977678571428573
```

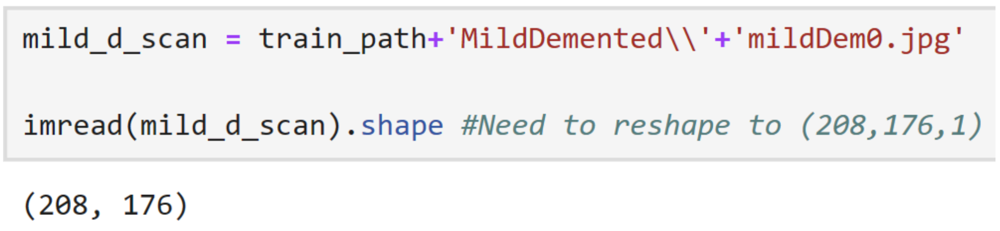
After determining the amount of images in the train and test sets, I realized that moderate demented images, and mild demented images would likely not be as accurate as the other images due to there being a significantly smaller sample size. I also determined that the model had a built in train/test split of approximately 80/20. Additionally, based on the largest class, non demented, I determined that the total amount of test images, 1279, divided by the number of non demented test images, 640, is .5004. This means that in order for the model to have any real value, it must be more accurate than this value.

Next, I wanted to view the shapes of the images, I selected one image at random.



```
os.listdir(train_path+'MildDemented')[0]
```

```
'mildDem0.jpg'
```

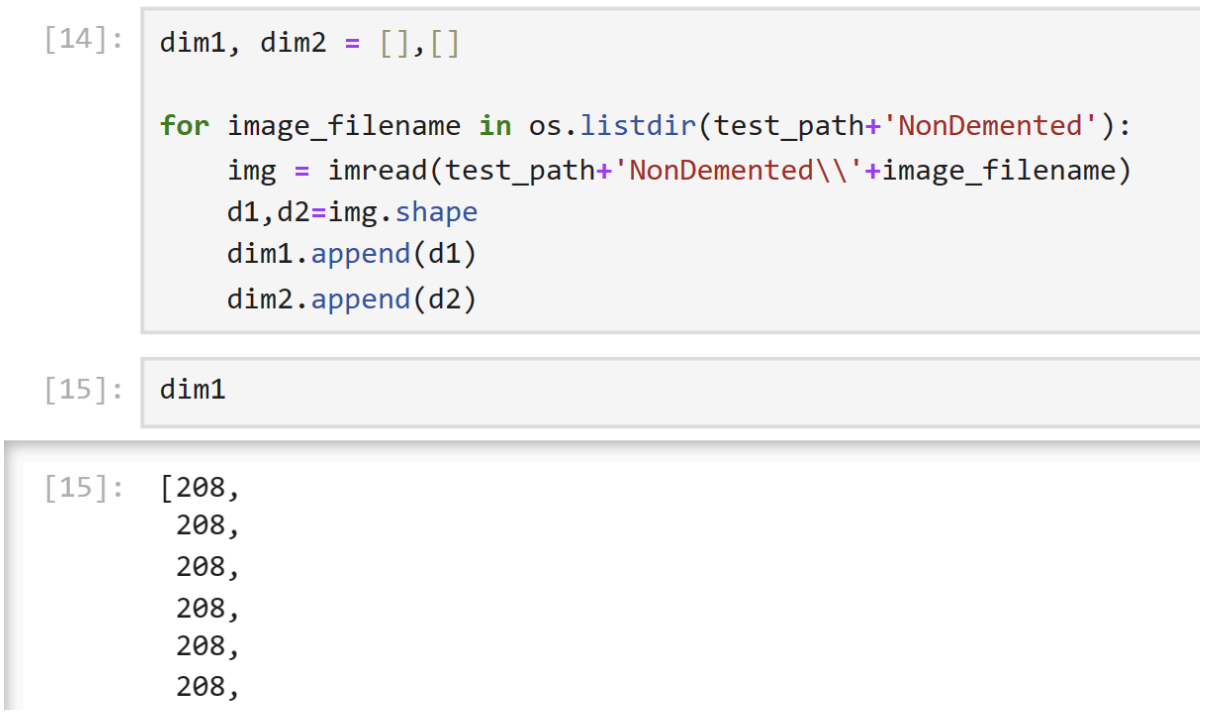


```
mild_d_scan = train_path+'MildDemented\\'+'mildDem0.jpg'

imread(mild_d_scan).shape #Need to reshape to (208,176,1)
```

```
(208, 176)
```

Since there was no specified color channel, I realized that I had to add a last element to the tuple to make the shape (208,176,1).

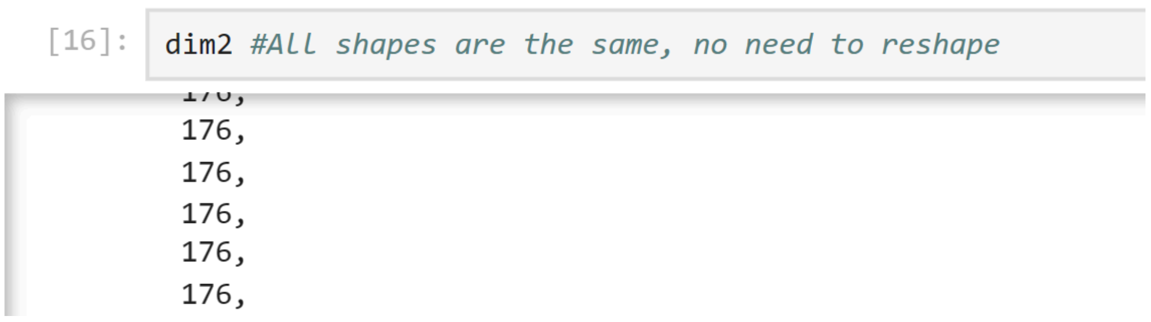
To ensure that all of the images were the same shape, I created a list of both the dimensions of the images in the whole list of non demented images.



```
[14]: dim1, dim2 = [],[]

      for image_filename in os.listdir(test_path+'NonDemented'):
          img = imread(test_path+'NonDemented\\'+image_filename)
          d1,d2=img.shape
          dim1.append(d1)
          dim2.append(d2)
```

```
[15]: dim1
```

```
[15]: [208,
       208,
       208,
       208,
       208,
       208,
```



```
[16]: dim2 #All shapes are the same, no need to reshape
```

```
       176,
       176,
       176,
       176,
       176,
```

Since the entire list was 208,176, we know we can use the same operation to update the shape of every single image in the dataset.

Next, I wanted to see the pixel values in the dataset to determine if they required normalization.

```
imread(vmild_d_scan).max()
```

```
243
```

```
imread(non_d_scan).max() #Need to normalize from a 1-255 scale
```

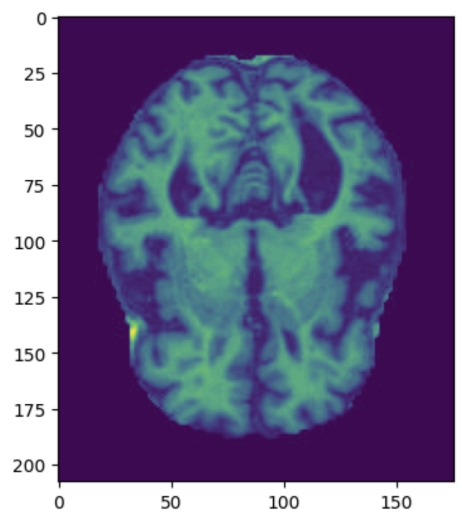
```
251
```

Since the greatest pixel values were near 255, we need to divide all of these values by 255.

Lastly, I plotted one image of every classification in order to see if I could pick up on any patterns:

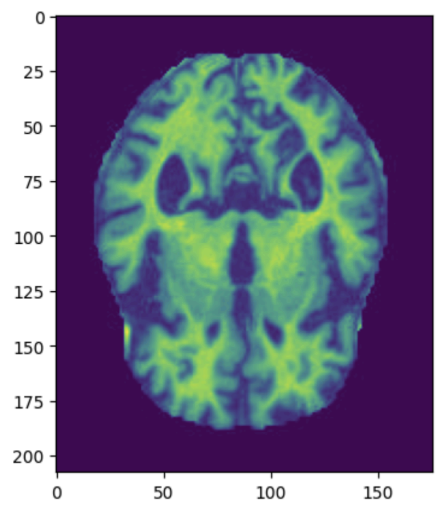
```
plt.imshow(imread(mild_d_scan)) #Random image of mild demented
```

```
<matplotlib.image.AxesImage at 0x210742c5240>
```



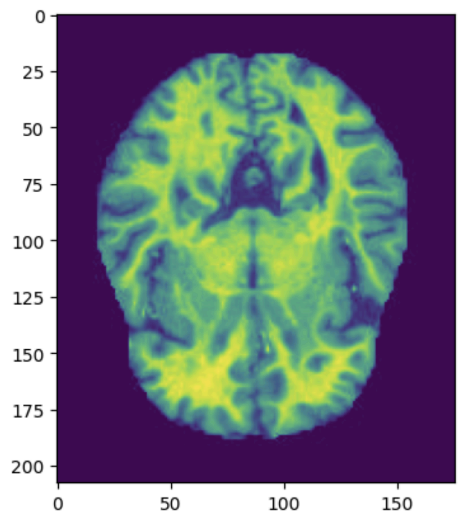
```
moderate_d_scan = train_path+'ModerateDemented\\'+'moderateDem0.jpg'
plt.imshow(imread(moderate_d_scan))
```

```
<matplotlib.image.AxesImage at 0x2107438f8b0>
```



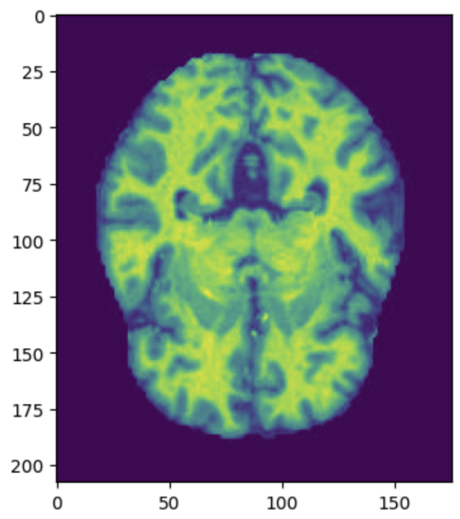
```
non_d_scan = train_path+'NonDemented\\'+'nonDem0.jpg'
plt.imshow(imread(non_d_scan))
```

```
<matplotlib.image.AxesImage at 0x21074b63df0>
```



```
vmild_d_scan = train_path+'VeryMildDemented\\'+'veryMildDem0.jpg'
plt.imshow(imread(vmild_d_scan))
```

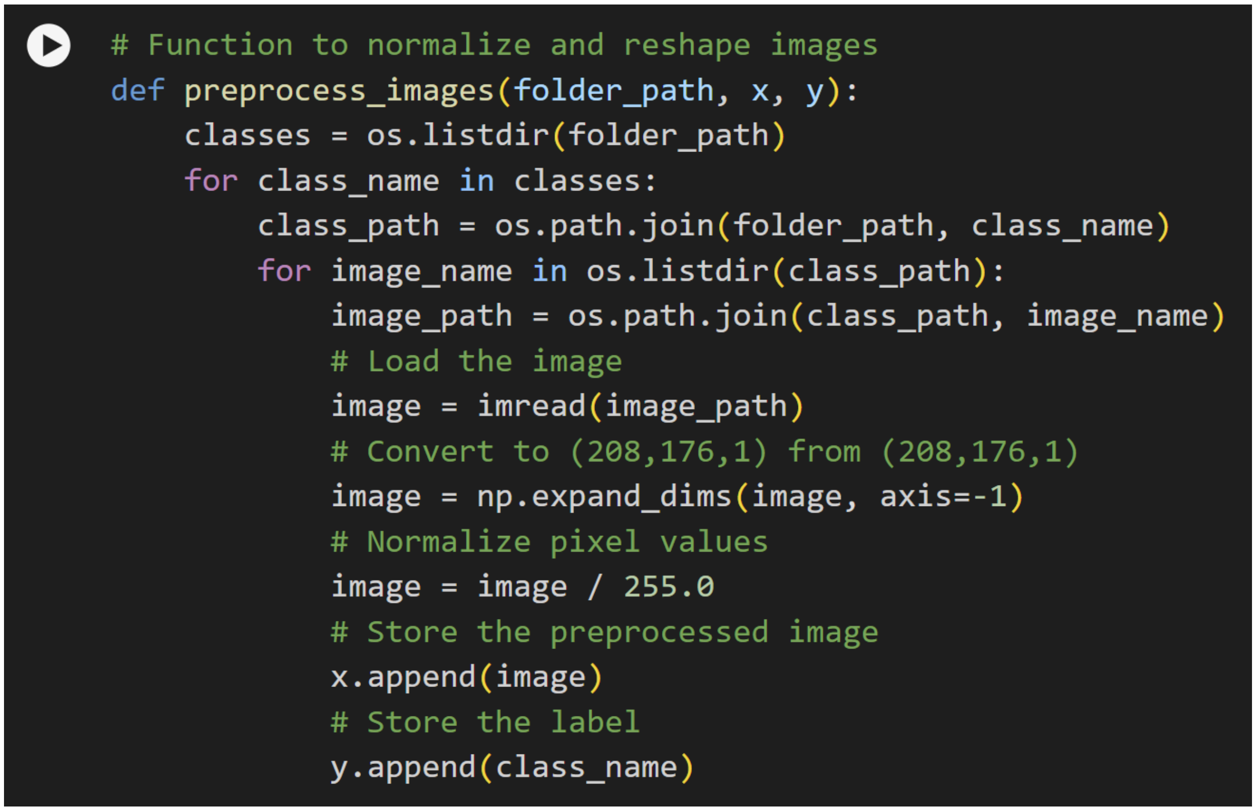
```
<matplotlib.image.AxesImage at 0x21074bef160>
```



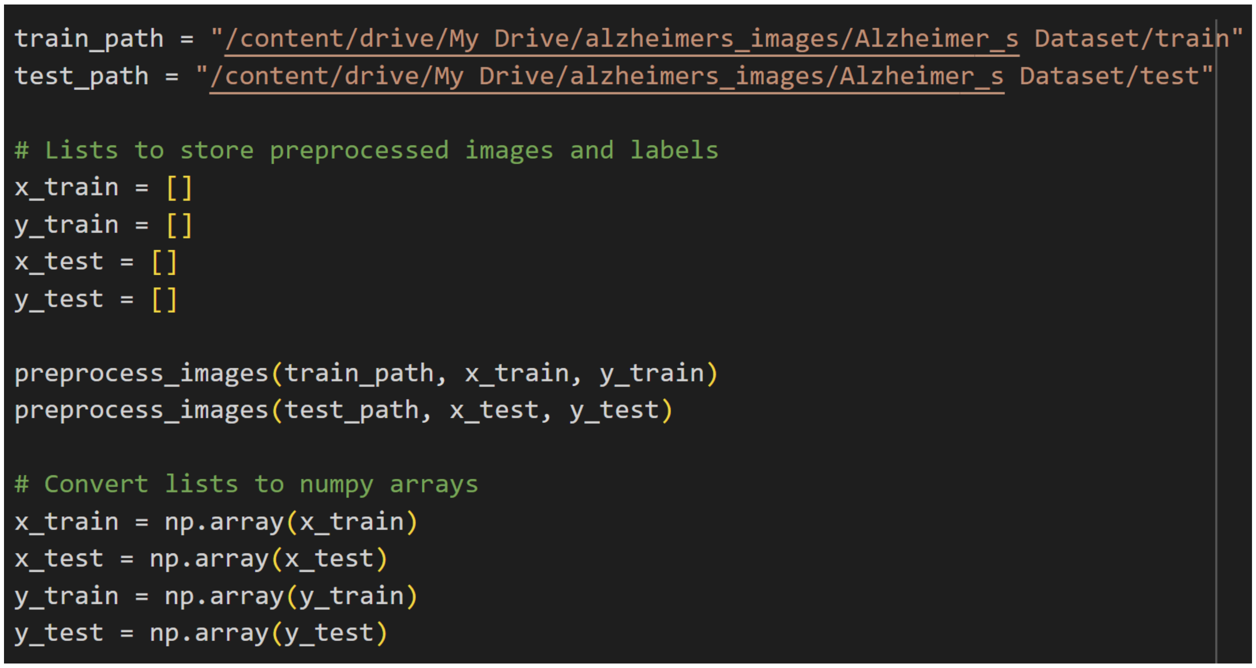
Through these images, the only things that I could pick up on was that there was less brightness the more significant the alzheimers became. This means that reshaping the data is likely not very necessary.

Step 3: Data Preprocessing

First, I created a function to read in the folder path, and arrays which stored the x values of the images for training and test data in array format, and the y values of the images for training and test data which were the correlating classifications of the images. This function transforms the image to fit the machine learning model by adding one color channel for a grayscale image. It then normalizes the pixel values of the arrays. It stores the image pixel values to x, and the class name to y. After passing the training data and test data through this function, I convert all of these lists of data to numpy arrays for compatibility with machine learning algorithms.

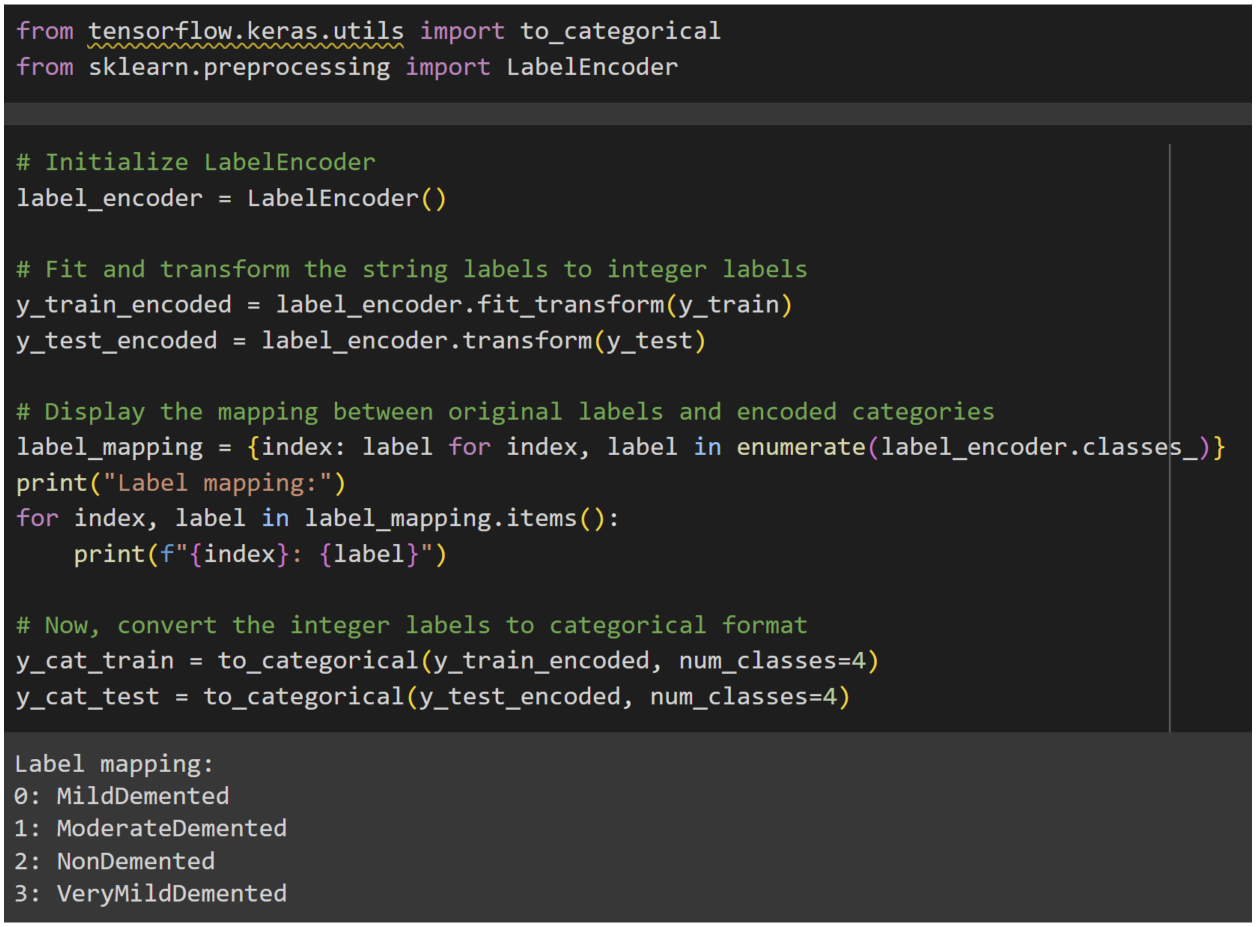


```
# Function to normalize and reshape images
def preprocess_images(folder_path, x, y):
    classes = os.listdir(folder_path)
    for class_name in classes:
        class_path = os.path.join(folder_path, class_name)
        for image_name in os.listdir(class_path):
            image_path = os.path.join(class_path, image_name)
            # Load the image
            image = imread(image_path)
            # Convert to (208,176,1) from (208,176,1)
            image = np.expand_dims(image, axis=-1)
            # Normalize pixel values
            image = image / 255.0
            # Store the preprocessed image
            x.append(image)
            # Store the label
            y.append(class_name)
```



```
train_path = "/content/drive/My Drive/alzheimers_images/Alzheimer_s Dataset/train"
test_path = "/content/drive/My Drive/alzheimers_images/Alzheimer_s Dataset/test"

# Lists to store preprocessed images and labels
x_train = []
y_train = []
x_test = []
y_test = []

preprocess_images(train_path, x_train, y_train)
preprocess_images(test_path, x_test, y_test)

# Convert lists to numpy arrays
x_train = np.array(x_train)
x_test = np.array(x_test)
y_train = np.array(y_train)
y_test = np.array(y_test)
```

Next, I needed to convert all of the y values to categorical by first mapping the category values to integers(which are displayed below), and then converting those integers to categorical format for compatibility with the neural network.



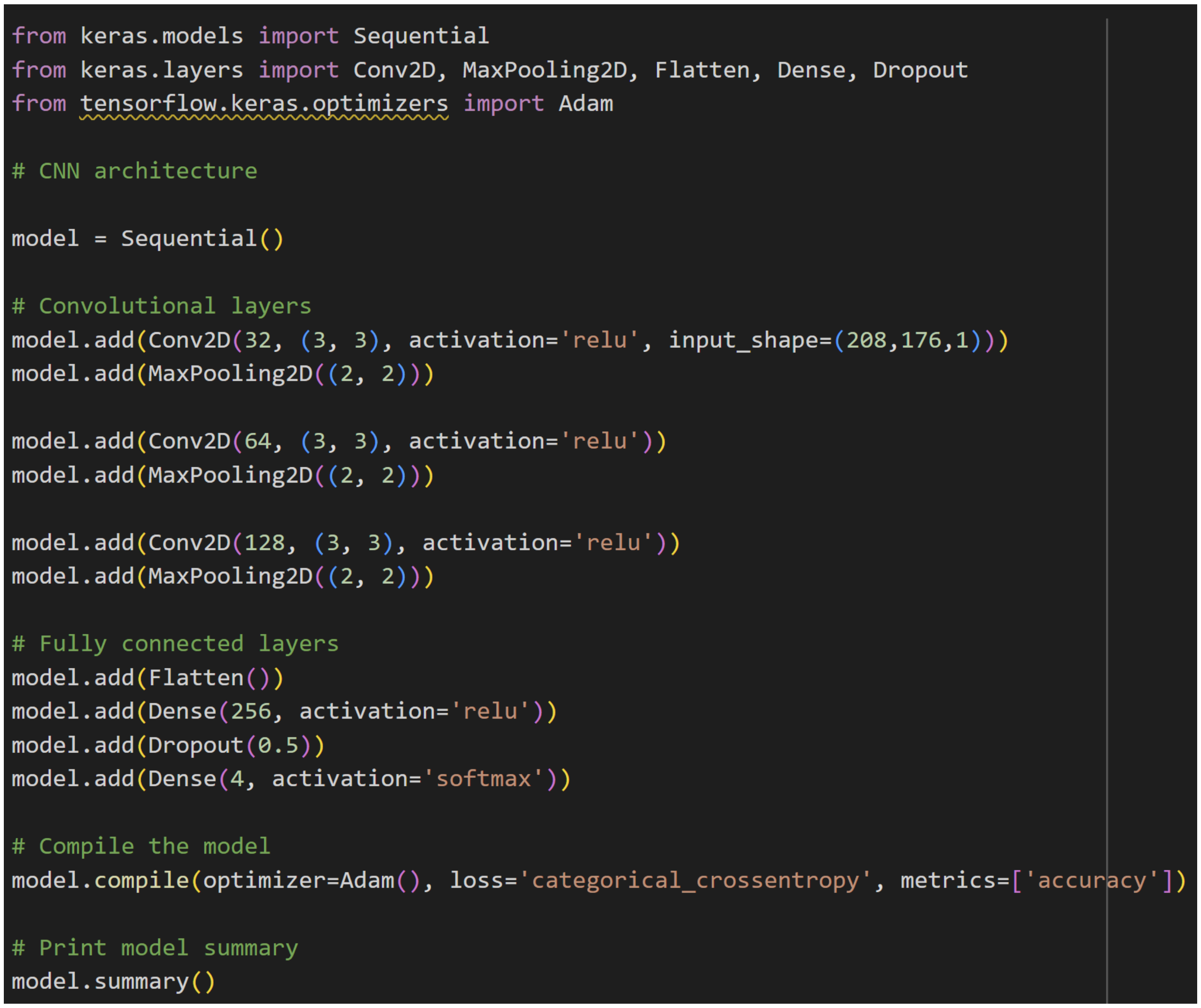
```python
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import LabelEncoder
```

```python
# Initialize LabelEncoder
label_encoder = LabelEncoder()

# Fit and transform the string labels to integer labels
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)

# Display the mapping between original labels and encoded categories
label_mapping = {index: label for index, label in enumerate(label_encoder.classes_)}
print("Label mapping:")
for index, label in label_mapping.items():
    print(f"{index}: {label}")

# Now, convert the integer labels to categorical format
y_cat_train = to_categorical(y_train_encoded, num_classes=4)
y_cat_test = to_categorical(y_test_encoded, num_classes=4)
```

```
Label mapping:
0: MildDemented
1: ModerateDemented
2: NonDemented
3: VeryMildDemented
```

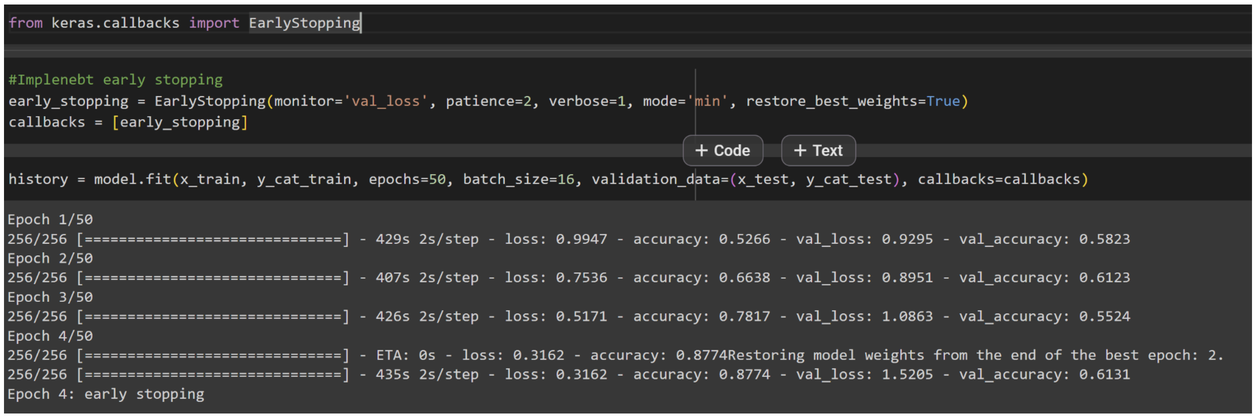
Step 4: Training the Model

I then set up the convolutional neural network architecture by importing Sequential, Conv2D, MaxPooling2D, Flatten, Dense, and Dropout. I created many layers including 3 convolutional layers, 3 max pooling layers, a flattening layer, a two dense layers, including the output layer, and a dropout layer. I ensured the output layer had 4 neurons, for the 4 categories. The model used categorical crossentropy to measure loss, and used the Adam optimizer.



```python
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam

# CNN architecture

model = Sequential()

# Convolutional layers
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(208,176,1)))
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))

# Fully connected layers
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4, activation='softmax'))

# Compile the model
model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])

# Print model summary
model.summary()
```

Lastly, I implemented early stopping, and fit the model to the training data, and validation data.



```python
from keras.callbacks import EarlyStopping

#Implenebt early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=2, verbose=1, mode='min', restore_best_weights=True)
callbacks = [early_stopping]

history = model.fit(x_train, y_cat_train, epochs=50, batch_size=16, validation_data=(x_test, y_cat_test), callbacks=callbacks)
```

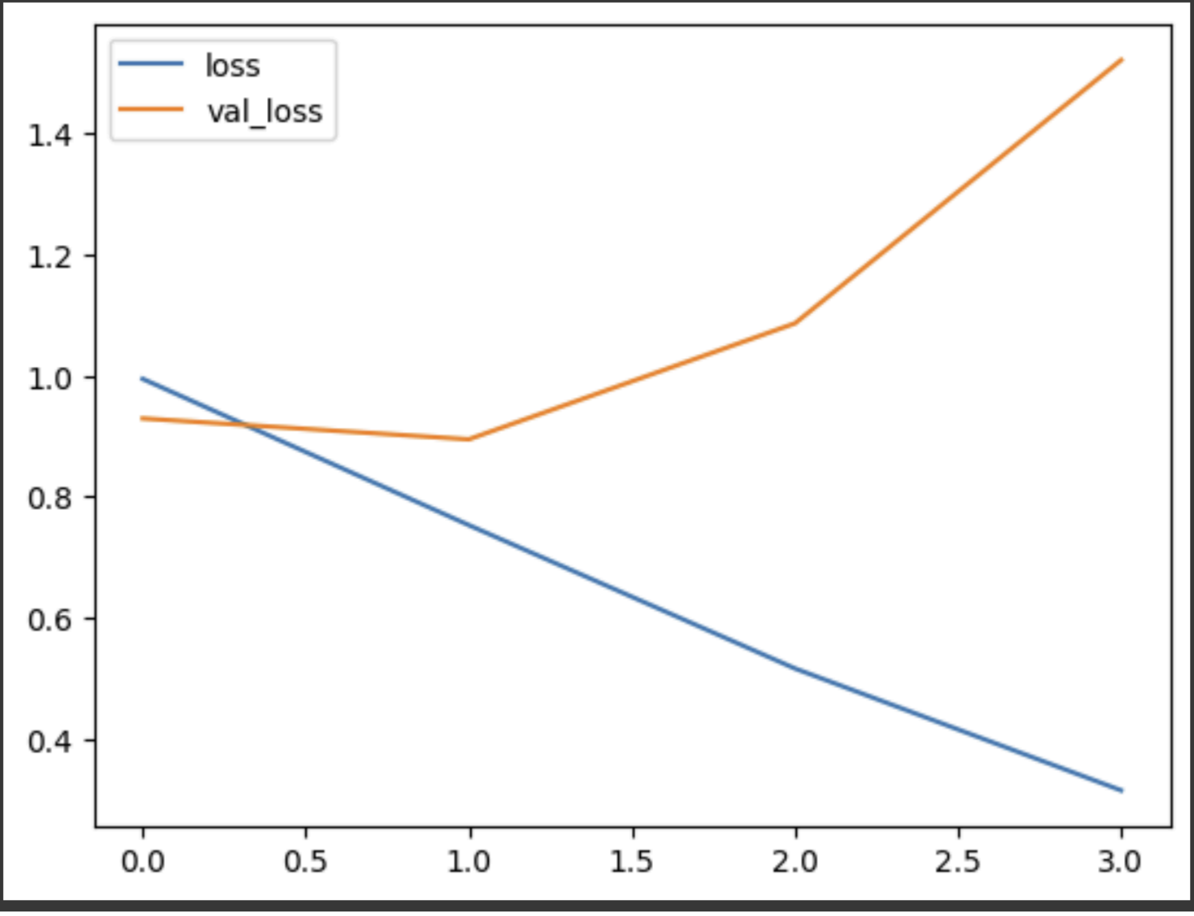
```
Epoch 1/50
256/256 [==============================] - 429s 2s/step - loss: 0.9947 - accuracy: 0.5266 - val_loss: 0.9295 - val_accuracy: 0.5823
Epoch 2/50
256/256 [==============================] - 407s 2s/step - loss: 0.7536 - accuracy: 0.6638 - val_loss: 0.8951 - val_accuracy: 0.6123
Epoch 3/50
256/256 [==============================] - 426s 2s/step - loss: 0.5171 - accuracy: 0.7817 - val_loss: 1.0863 - val_accuracy: 0.5524
Epoch 4/50
256/256 [==============================] - ETA: 0s - loss: 0.3162 - accuracy: 0.8774Restoring model weights from the end of the best epoch: 2.
256/256 [==============================] - 435s 2s/step - loss: 0.3162 - accuracy: 0.8774 - val_loss: 1.5205 - val_accuracy: 0.6131
Epoch 4: early stopping
```

Step 5: Analyzing the Results

```
metrics = pd.DataFrame(model.history.history)
```

```
metrics[['loss','val_loss']].plot()
```

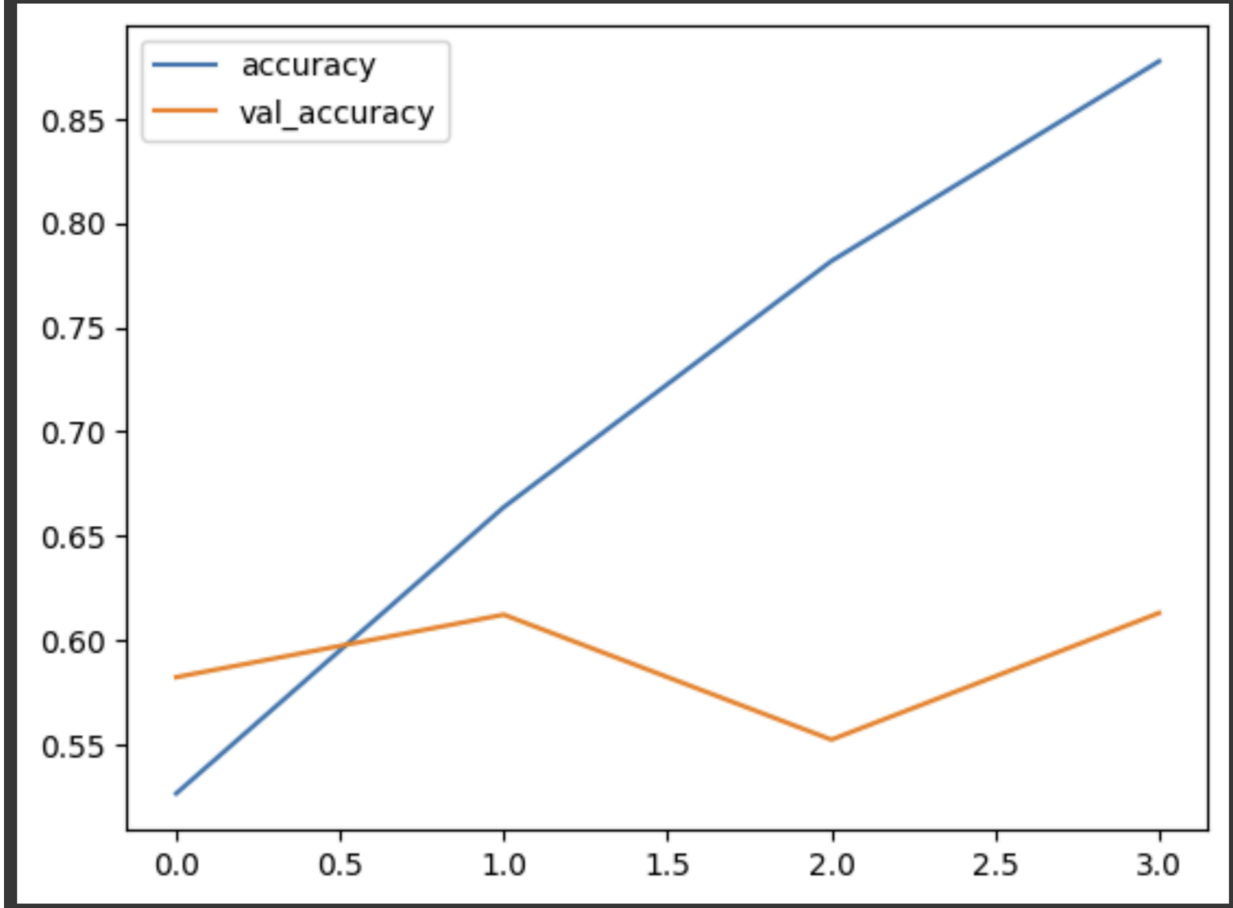
```
<Axes: >
```



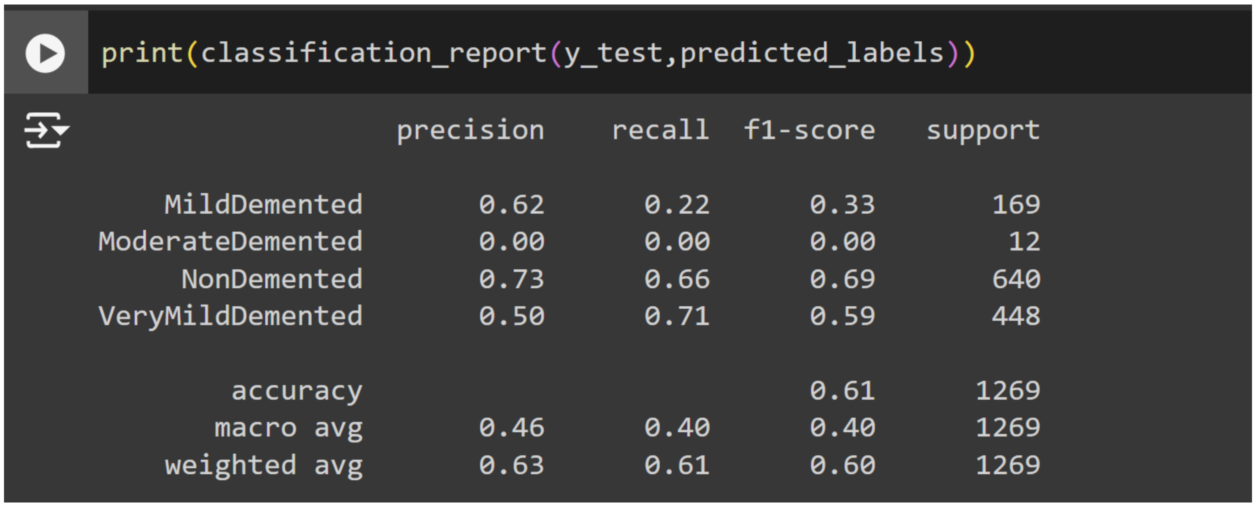
We can see that the model overfitted to the training data since the training loss decreased overtime, while the validation loss increased over time. This is likely due to a lack of data in some of the categories, which leads to the same few images being overfitted to. The main way that I attempted to combat this was through early stopping and restoring the best weights.

```
metrics[['accuracy','val_accuracy']].plot()
```

```
<Axes: >
```

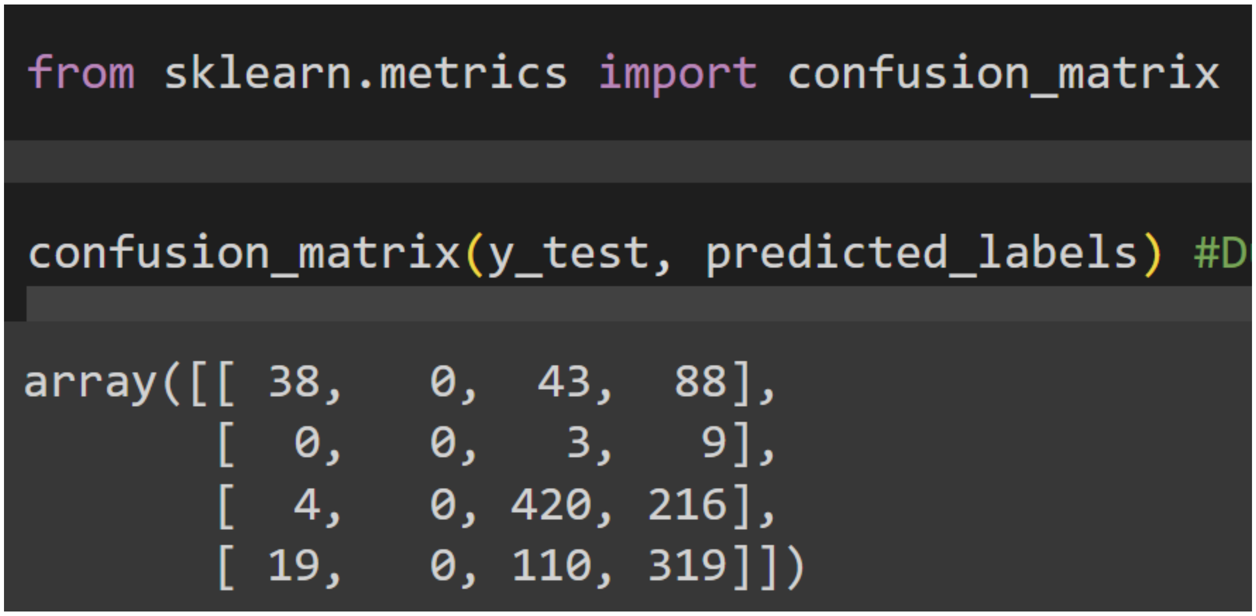


We can again see the effects of overfitting of the model due to a significantly higher accuracy of training data as compared to validation accuracy. I have not viewed all of the images in the training data, but I think this may be due to the validation images and training images not having many similarities. This assumption is not only based off of my training, but off of the other kaggle codes that I viewed, where the greatest accuracy any TensorFlow user achieved was .56.



```
print(classification_report(y_test,predicted_labels))

                  precision    recall  f1-score   support

    MildDemented       0.62      0.22      0.33       169
ModerateDemented       0.00      0.00      0.00        12
     NonDemented       0.73      0.66      0.69       640
VeryMildDemented       0.50      0.71      0.59       448

        accuracy                           0.61      1269
       macro avg       0.46      0.40      0.40      1269
    weighted avg       0.63      0.61      0.60      1269
```

We can see that as compared to the .56 accuracy of other users, my training data was able to achieve an accuracy of .61. Additionally, it is worth noting that the MildDemented and ModerateDemented classes had very low f1 scores due to a scarcity of data.



```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_test, predicted_labels) #D

array([[ 38,   0,  43,  88],
       [  0,   0,   3,   9],
       [  4,   0, 420, 216],
       [ 19,   0, 110, 319]])
```

We see that the majority of predictions for all of these groups falls into NonDemented and VeryMildDemented which is not surprising since there was a lot more data for these two groups, leading to overfitting. It is interesting however, that these two groups were still more distinguishable from each other than random guesses. 420/640 NonDemented images were correctly identified, and 319/448 VeryMildDemented images were identified, while only 38/169 MildDemented images were correctly identified, and 0 out of 12 ModerateDemented images were identified. This is likely due to the model getting negative feedback whenever ModerateDemented was the predicted class since it is only accurate 12/1269 times (because of the scarcity of the data), resulting in 0 total predictions of ModerateDemented images.

Additionally, MildDemented images were frequently mistaken for VeryMildDemented images, and much less frequently mistaken for NonDemented images, and same vice versa, that VeryMildDemented images were quite frequently mistaken for MildDemented images. NonDemented images were almost never mistaken for MildDemented, and were never mistaken for ModerateDemented images, meaning that if the model found a NonDemented image, it was very accurate in determining that there was an extremely low level of Alzheimers(either NonDemented or VeryMildDemented) in the patient.