Alzheimer's Classification

Development NN for detection and classification of Alzheimer's disease.

Authors:
Martin Martuccio
Samuele Pellegrini

Introduction

1

Context

Alzheimer's is a neurodegenerative disorder causing memory loss and cognitive decline.

2

Early Diagnosis:

Early diagnosis improves treatment outcomes and quality of life for quality of life for Alzheimer's patients.

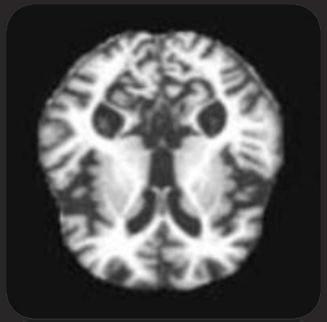
3

Role of Machine Learning:

Machine learning aids in early detection and classification of Alzheimer's disease.

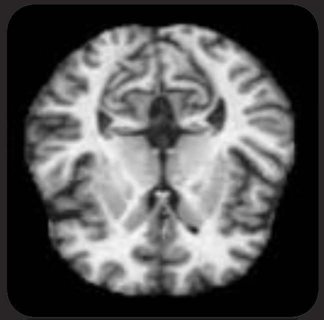
Project Overview

In this project, we aim to develop a Convolutional Neural Network (CNN) model to accurately classify Magnetic Resonance Imaging (MRI) scans into the four distinct stages of Alzheimer's disease.



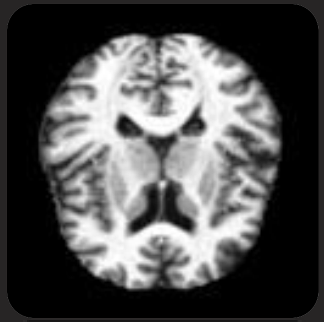
Mild Demented

The dataset utilized in this project was sourced from Kaggle.



Very Moderate Demented

Moderate Demented



No Demented

Tools and Framework

Leveraging TensorFlow and Keras for machine learning and deep learning.
Using Google Colab for a collaborative and GPU-accelerated environment.

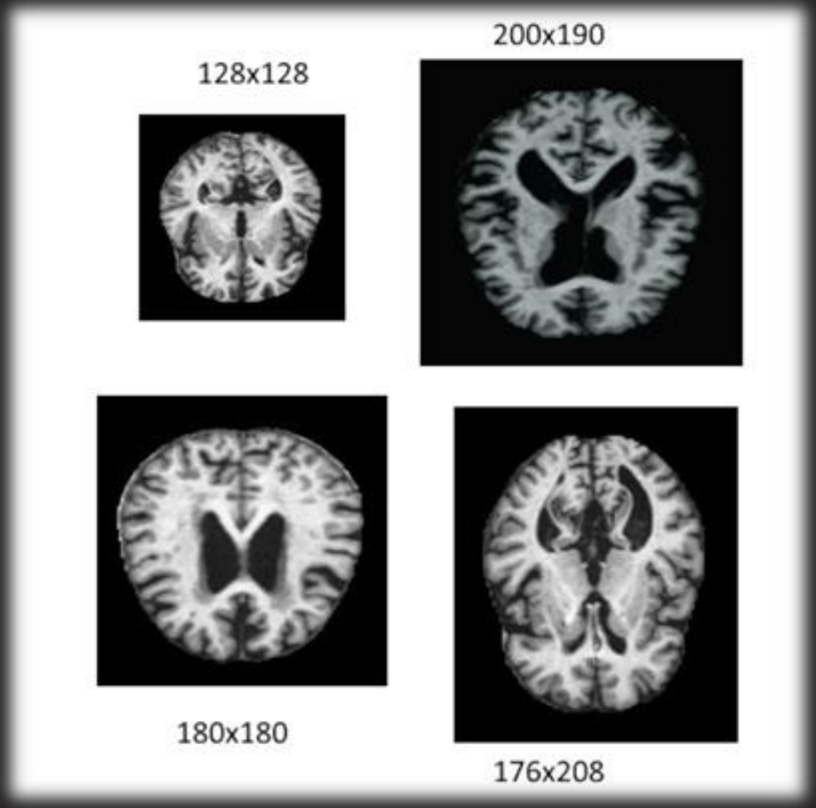
Google Colab

TensorFlow and Keras

Google Colab

TensorFlow

Data Preparation



1.Initial Analysis

Script's results to count the number of images and their dimensions.
The data obtained:

1. *176x208*: **6400** images
2. *180x180*: **6400** images
3. *128x128*: **6400** images
4. *200x190*: **27122** images

2.Image Resizing

All images were resized to 128x128 pixels to ensure uniformity

3.Importance

Uniform image dimensions are crucial for the neural network's performance as they ensure consistent input size, leading to better training efficiency and accuracy.

Data Augmentation

Importance:

•Data augmentation is important for handling **imbalanced datasets** and improving model **robustness**. It helps in increasing the diversity of the training data, which enhances the model's ability to **generalize**.

Techniques Used:

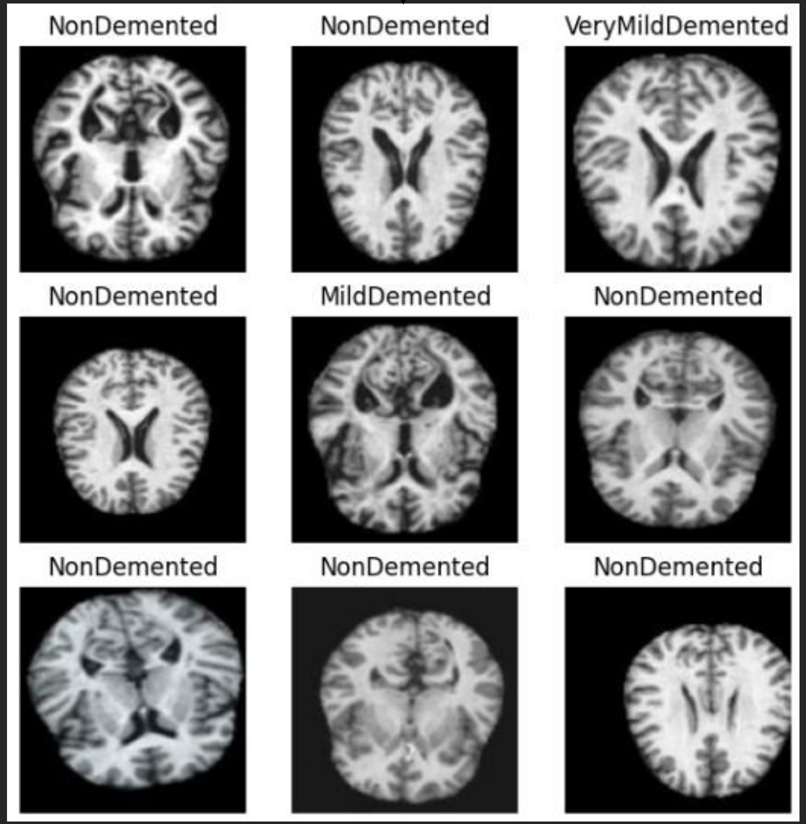
•**Rotation**: to simulate different viewing angles.

•**Zoom**: to simulate different distances.

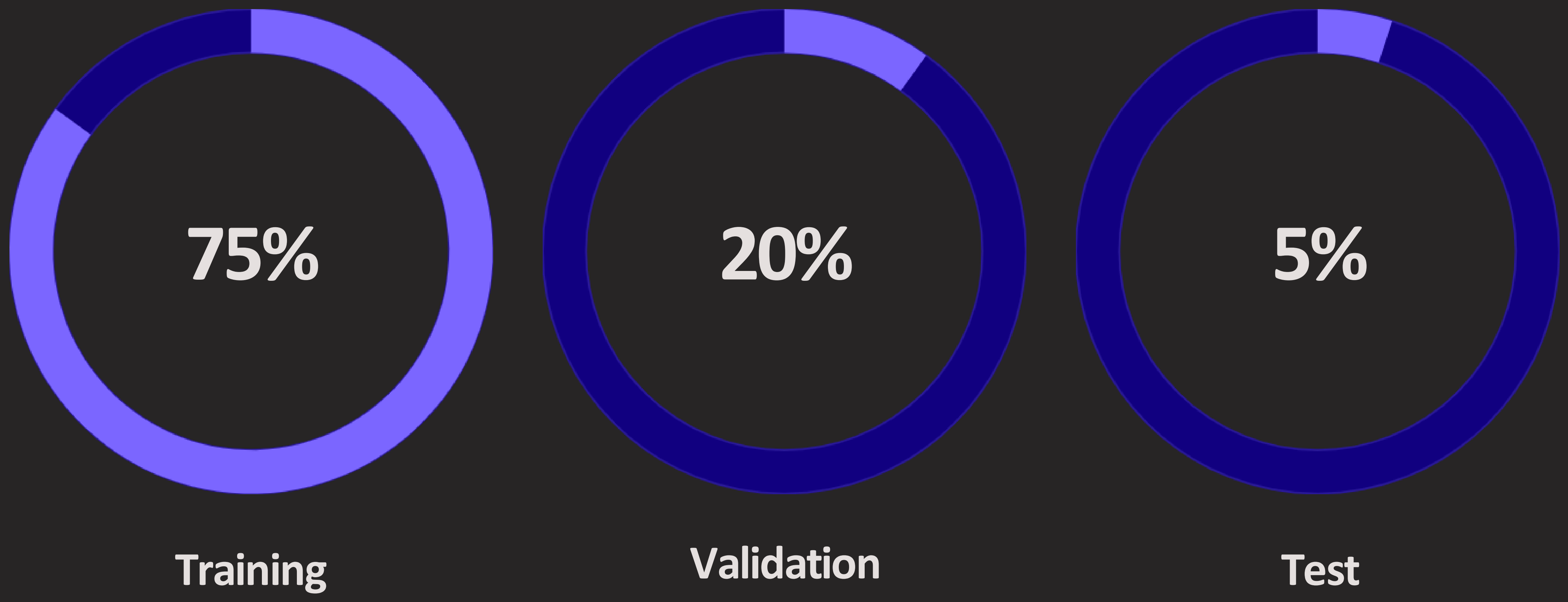
•**Translation**: to simulate different positions.

•**Flipping**: to create mirror images.

```python
# Define the preprocessing and data augmentation layers externally
rescale = tf.keras.Sequential([
    layers.Rescaling(1./255)
])

data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal_and_vertical"),
    layers.RandomRotation(0.2, "nearest"),
    layers.RandomZoom(0.15, 0.15, "nearest")
])

data_augmentation1 = tf.keras.Sequential([
    layers.RandomRotation(0.2, "nearest")
])

data_augmentation2 = tf.keras.Sequential([
    layers.RandomZoom(0.15, 0.15, "nearest")
])

data_augmentation3 = tf.keras.Sequential([
    layers.RandomTranslation(0.15, 0.15, "nearest")
])
```



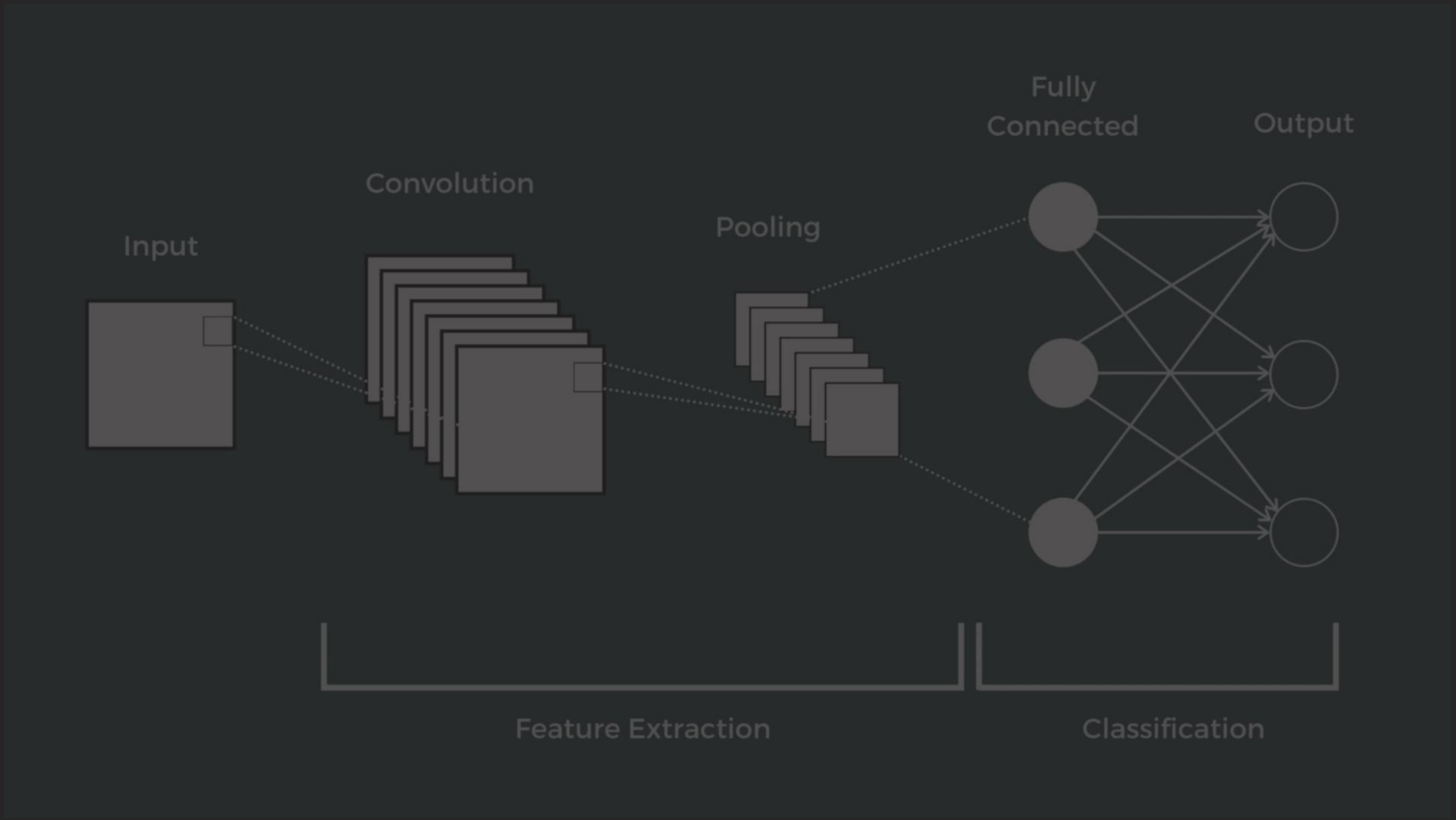
Splitting the dataset



We split the dataset into three subsets: **75%** for **training**, **20%** for **validation**, and **5%** for **testing**.

Model Development

In our approach to developing models for classifying Alzheimer's stages, we evaluated **12 different neural network architectures**. We referenced research papers, implemented custom models, and used TensorFlow documentation to explore various layers, including a ResNet.



ORIGINAL RESEARCH article

Front. Public Health, 07 February 2022
Sec. Digital Public Health
Volume 10 - 2022 |
https://doi.org/10.3389/fpubh.2022.834032

This article is part of the Research Topic
Big Data Analytics for Smart Healthcare applications
View all 109 articles

Evaluation of Neuro Images for the Diagnosis of Alzheimer's Disease Using Deep Learning Neural Network

Ahila A[1*] Poongodi M[2*] Mounir Hamdi[2] Sami Bourouis[3] Kulhanek Rastislav[4] Faizaan Mohmed[5]


```
# Original
class CustomNN(Model):
    def __init__(self):
        super(CustomNN, self).__init__()
        # Definizione dei layer
        self.conv1 = Conv2D(16, (3, 3), activation='relu', padding='same', input_shape=(128, 128, 1))
         # The first convolutional layer with 16 filters, 3x3 kernel size, ReLU activation, and same padding
        self.pool1 = MaxPooling2D((2, 2), strides=(2, 2))
        self.conv2 = Conv2D(32, (3, 3), activation='relu', padding='same')
        self.pool2 = MaxPooling2D((2, 2), strides=(2, 2))
        self.conv3 = Conv2D(64, (3, 3), activation='relu', strides=(2, 2))
        self.flatten = Flatten()
        self.fc1 = Dense(512, activation='relu')
        self.fc2 = Dense(4, activation='softmax') # The second fully connected layer with softmax activation (for classification)

    def call(self, x):
         # Defining the input flow through the layers
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.conv3(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.fc2(x)
        return x

    # from papar: https://www.frontiersin.org/journals/public-health/articles/10.3389/fpubh.2022.834032/full
```

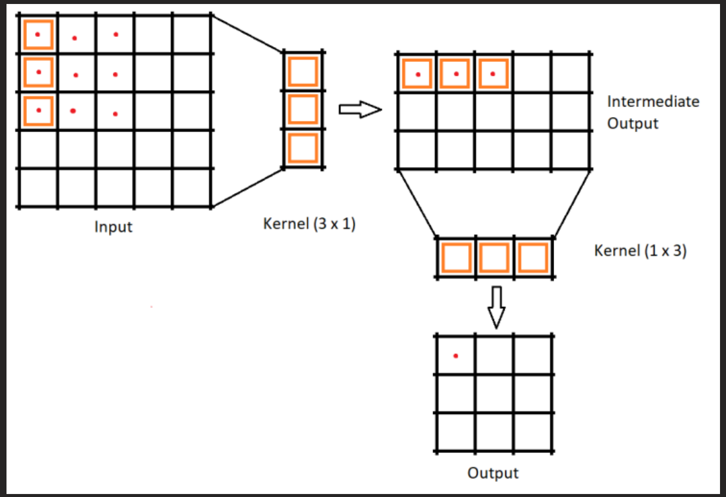
Referral links: Paper 1°, Paper 2°, Paper 3°

Custom Implementations and Exploring TensorFlow layers

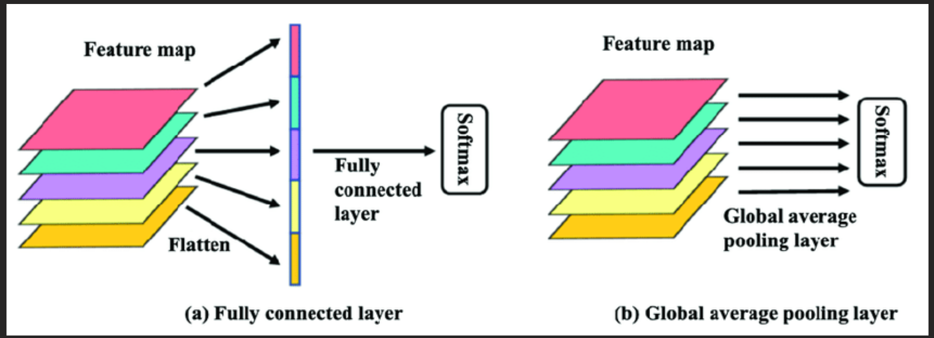
We designed several custom models to test different configurations and layers, including convolutional, pooling, and dense layers.

We explored TensorFlow documentation to use layers such as SeparableConv2D, GlobalAveragePooling2D, and Batch Normalization.

SeparableConv2D



GlobalAvaragePooling2D



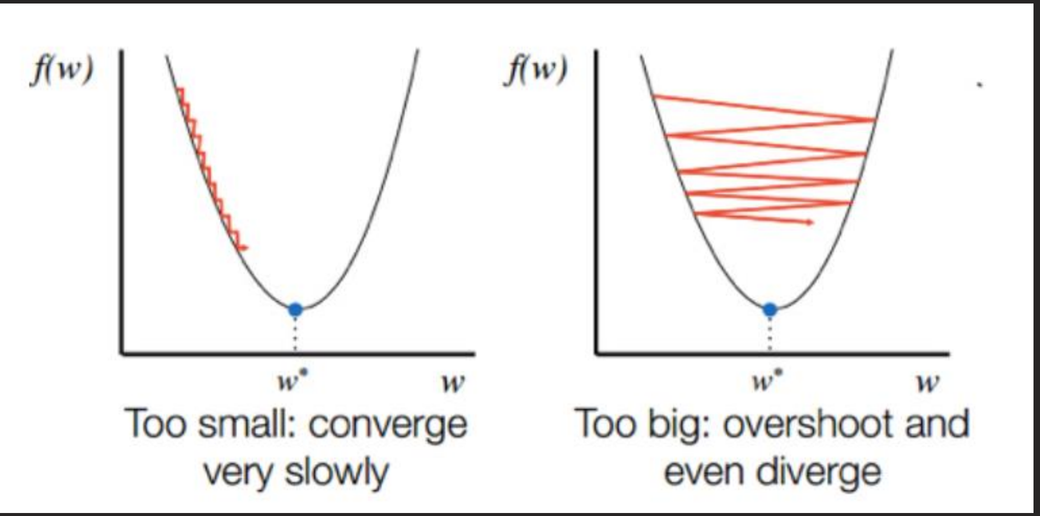
(a) Fully connected layer (b) Global average pooling layer

```
class CustomNN6(Model):
    def __init__(self):
        super(CustomNN6, self).__init__()
        self.conv1 = Conv2D(4, (3, 3), activation='relu', padding='same'
        self.pool1 = MaxPooling2D((3, 3), strides=(2, 2))
        self.conv2 = Conv2D(8, (3, 3), activation='relu', padding='same'
        self.pool2 = MaxPooling2D((3, 3), strides=(2, 2))
        self.conv3 = Conv2D(16, (3, 3), activation='relu', padding='same
        self.pool3 = MaxPooling2D((2, 2), strides=(2, 2))
        self.conv4 = Conv2D(32, (3, 3), activation='relu', padding='same
        self.pool4 = MaxPooling2D((2, 2), strides=(2, 2))
        self.flatten = Flatten()
        self.bn = BatchNormalization()
        self.fc1 = Dense(256, activation='leaky_relu')
        self.drop1 = Dropout(0.45) # Drop some nodes dur
        self.fc2 = Dense(8, activation='relu')
        self.drop2 = Dropout(0.4) # Drop some nodes duri
        self.fc3 = Dense(4, activation='softmax')
```

```
class CustomNN8(Model):
    def __init__(self):
        super(CustomNN8, self).__init__()
        self.conv1 = SeparableConv2D(32, (3, 3), padding='same', input_shape=(128, 128, 1))
        self.bn1 = BatchNormalization()
        self.pool1 = MaxPooling2D((2, 2), strides=(2, 2))
        self.conv2 = SeparableConv2D(64, (3, 3), padding='same')
        self.bn2 = BatchNormalization()
        self.pool2 = MaxPooling2D((2, 2), strides=(2, 2))
        self.conv3 = SeparableConv2D(128, (3, 3), strides=(2, 2))
        self.bn3 = BatchNormalization()
        self.pool3 = MaxPooling2D((2, 2), strides=(2, 2))
        self.flatten = Flatten()
        self.fc1 = Dense(256, activation='relu')
        self.dropout = Dropout(0.5)
        self.fc2 = Dense(4, activation='softmax')
```

```
class CustomNN10(Model):
    def __init__(self):
        super(CustomNN10, self).__init__()
        self.conv1 = Conv2D(32, (3, 3), padding='same', input_shape=(128, 128, 1))
        self.bn1 = BatchNormalization()
        self.conv2 = Conv2D(32, (3, 3), padding='same')
        self.bn2 = BatchNormalization()
        self.add = Add()
        self.pool1 = MaxPooling2D((2, 2), strides=(2, 2))
        self.conv3 = Conv2D(64, (3, 3), padding='same')
        self.bn3 = BatchNormalization()
        self.conv4 = Conv2D(64, (3, 3), padding='same')
        self.bn4 = BatchNormalization()
        #self.add2 = Add()
        self.global_pool = GlobalAveragePooling2D()
        self.fc1 = Dense(128, activation='relu')
        self.dropout = Dropout(0.5)
        self.fc2 = Dense(4, activation='softmax')
```

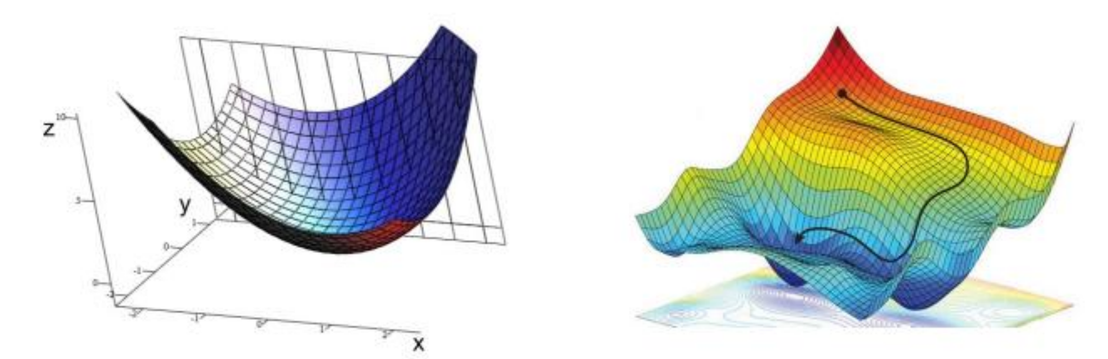
Learning Rate and Loss Functions



Learning Rate

The learning rate controls how much to change the model in response to the estimated error each time the model weights are updated.

We used a learning rate of 0.001 which is best suited for most optimizers.



Loss Functions

These functions measure the error between the predicted and actual labels.

- Two types of loss functions:

  1° Sparse Categorical Crossentropy

  2° Kullback-Leibler Divergence

$$KL(P||Q) = \frac{\sum_x P(x) log \frac{P(x)}{Q(x)}}{n}$$

How one probability distribution diverges from a second, expected probability distribution.
P(x) the true probability
Q(x) the predicted probability

$$L = -\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log(\hat{y}_i)$$

It converts the true labels into **one-hot encoded vectors** internally and then applying the regular categorical cross-entropy loss calculation.

color
red
green
blue
red

color
0
1
2
0

Model Comparison and Selection

After training all models, we compared their performance based on **accuracy** and **loss values** at different stages. Each model was evaluated with various configurations. To determine the best models, we looked at the values from the last iteration and the maximum values during training. The top-performing models were selected for further optimization.

Legend Explanation:

Model Identifiers: Each model has unique configurations and architectures.

M	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11
1° Paper	M + CVL	2° Paper	M - CVL	M2: 8 CVL	Martin 1	Samuele 1	Martin 2	Martin 3	Martin 4	Martin 5	ResNet50

O1	O2	N	A
sparse categ. crossen.	kullback leibler div.	Training without DA	Training with DA

Training Strategies (N, A)

Loss Functions (O1, O2)
- **O1:** Sparse Categorical Cross-Entropy
- **O2:** Kullback-Leibler Divergence

Last Iteration: Accuracy and loss at the final training iteration.

Max (Acc) & Min (Loss): Peak accuracy and lowest loss during training.

Data Values	Last Iteration						Max (Acc) & Min (Loss)			
	Training		Validation		Testing		Training		Validation	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss

Ordered tables

Ordered By Training	
O1_N	M11
	M
	M3
	M1
	M4
	M5
	M2
	M7
	M8
	M6
	M10
	M9
O1_A	M1
	M
	M3
	M4
	M2
	M5
	M6
	M7
	M8
	M9
	M10
	M11

Ordered By Validation	
O1_N	M1
	M4
	M2
	M5
	M7
	M3
	M
	M8
	M6
	M9
	M11
	M10
O1_A	M4
	M
	M2
	M1
	M3
	M6
	M7
	M5
	M8
	M9
	M10
	M11

Parameter Optimization Overview

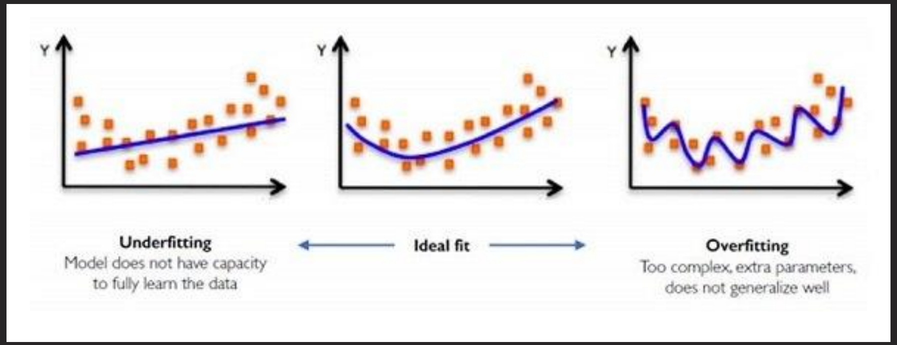
How various parameters influence model performance ?

Through a detailed analysis of performance metrics, we identified **the optimal configurations** for each model. A color-coded table clearly highlights trends and help us to discard the worst models.

Useful to spot …

Underfitting occurs when the model is too simplistic to capture the underlying patterns in the data.

Overfitting indicates that the model performs exceptionally well on the training data, but fails to generalize to new, unseen data.



M	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	Check Scripts for Details
1° Paper	M + CVL	2° Paper	M - CVL	M2: 8 CVL	Martin 1	Samuele 1	Martin 2	Martin 3	Martin 4	Martin 5	ResNet50	

O1	O2	N	A
sparse categ. crossen.	kullback leibler div.	Training without DA	Training with DA

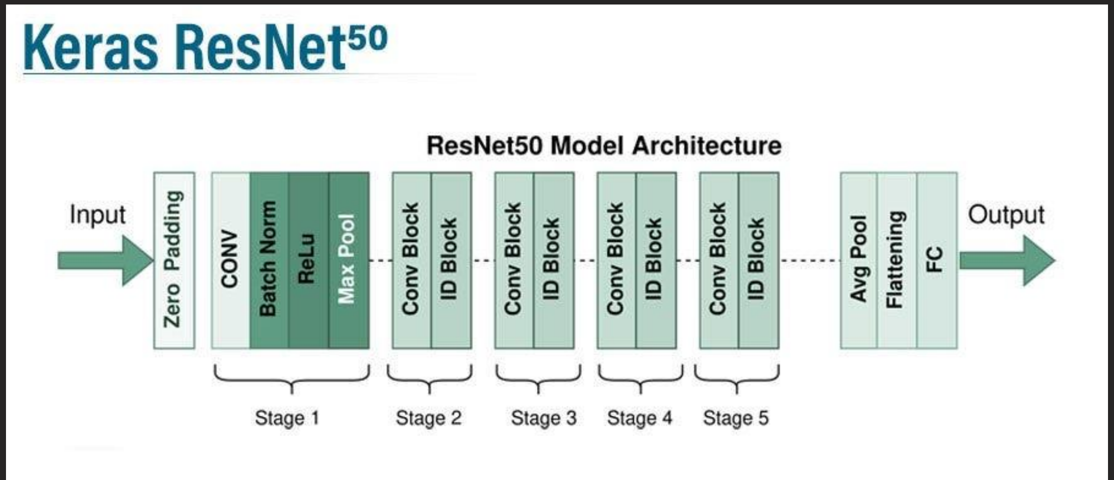
Data Values		Last Iteration						Max (Acc) & Min (Loss)			
		Training		Validation		Testing		Training		Validation	
		Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
O1_N	M	0,970813	0,086918	0,840278	0,942821	0,831325	0,92084	0,970813	0,086918	0,849175	0,477502
	M1	0,954723	0,123859	0,902127	0,353786	0,903184	0,358114	0,954723	0,123859	0,903429	0,286309
	M2	0,924558	0,193682	0,892795	0,310255	0,888985	0,316839	0,924558	0,193682	0,892795	0,308043
	M3	0,970641	0,090223	0,851345	0,826116	0,844664	0,820539	0,970641	0,090223	0,853299	0,427012
	M4	0,940562	0,156737	0,900608	0,30994	0,90132	0,311124	0,940562	0,156737	0,911675	0,236602
	M5	0,938259	0,158715	0,892144	0,299664	0,88253	0,345283	0,938259	0,158715	0,897786	0,273791
	M6	0,610443	0,829959	0,568793	0,899124	0,561388	0,912741	0,610443	0,829959	0,64171	0,730231
	M7	0,907806	0,221938	0,881293	0,320778	0,873064	0,340033	0,907806	0,221938	0,897786	0,241308
	M8	0,676703	0,683147	0,828993	0,437899	0,831038	0,457268	0,676703	0,683147	0,829427	0,437899
	M9	0,345864	1,333494	0,33941	1,341201	0,346816	1,336558	0,346842	1,331396	0,33941	1,331101
	M10	0,479362	1,094499	0,212891	1,810457	0,212134	1,823538	0,481262	1,094499	0,417535	1,213668
	M11	0,980427	0,055931	0,336806	9535,638	0,345955	9355,894	0,980427	0,055931	0,338108	469,5905
O1_A	M	0,954033	0,119799	0,855469	101,5868	0,852123	99,51452	0,954033	0,119799	0,894531	52,80975
	M1	0,963358	0,096333	0,832899	138,9714	0,822576	139,6061	0,963358	0,096333	0,898438	46,87912
	M2	0,918283	0,204798	0,840712	63,11361	0,836919	62,37116	0,918283	0,204798	0,894314	30,77151
	M3	0,94036	0,152931	0,820095	87,96473	0,817269	89,20181	0,94036	0,152931	0,825955	74,01408
	M4	0,928127	0,186115	0,89822	47,09663	0,893574	47,72771	0,928127	0,186115	0,905816	34,50066
	M5	0,759484	0,551318	0,538845	226,1006	0,527395	226,2507	0,759484	0,551318	0,538845	73,03018
	M6	0,705716	0,63784	0,710069	45,6938	0,718158	43,58591	0,705716	0,63784	0,710069	12,84479
	M7	0,67843	0,679675	0,699653	56,1796	0,682588	55,75502	0,67843	0,679675	0,72526	13,27193
	M8	0,569656	0,918822	0,494141	33,4912	0,497705	32,48192	0,569656	0,918822	0,580295	9,361578
	M9	0,563151	0,929742	0,262153	223,3725	0,258032	229,8507	0,563151	0,929742	0,339193	50,8092
	M10	0,458062	1,147688	0,190972	416,9665	0,193632	424,1612	0,45953	1,146843	0,386068	236,9265
	M11	0,2636	4,279483	0,142795	37,2626	0,137694	37,0009	0,2636	4,279482	0,292752	27,12726
O2_N	M	0,291578	37,3172	0,288194	37,19965	0,280838	36,91074	0,291578	37,31562	0,291667	37,16817
	M1	0,228254	37,31723	0,23112	37,17867	0,235944	36,94543	0,228254	37,31714	0,232639	37,10519
	M2	0,291578	37,31724	0,288845	37,13668	0,279834	36,92461	0,291635	37,30269	0,292101	37,13668
	M3	0,291578	37,31723	0,291016	37,19966	0,280981	36,95234	0,291578	37,3172	0,291016	37,1157
	M4	0,256232	4,279462	0,272352	4,265982	0,260614	4,235236	0,264262	4,279458	0,323351	4,26117
	M5	0,228254	37,31722	0,23112	37,17867	0,236087	36,93849	0,228254	37,28795	0,231771	37,14717
	M6	0,227477	4,279462	0,229384	4,273203	0,236804	4,232053	0,277906	4,279462	0,346354	4,259965
	M7	0,228254	4,279463	0,231988	4,258763	0,237091	4,230462	0,256145	4,27946	0,335503	4,258762
	M8	0,228254	4,279465	0,229384	4,273203	0,235944	4,236824	0,262219	4,27946	0,288628	4,261168
	M9	0,149675	37,31722	0,141276	37,21014	0,136833	36,95929	0,324967	37,2941	0,336589	37,12619
	M10	0,219216	4,279465	0,229384	4,273201	0,237665	4,228079	0,274106	4,279456	0,338108	4,262371
	M11	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?
O2_A	M	0,241005	4,279461	0,25803	4,316164	0,259897	4,299351	0,255253	4,279459	0,26237	4,316164
	M1	0,263974	4,279462	0,26237	4,275856	0,25545	4,245823	0,273502	4,279459	0,274306	4,263356
	M2	0,25485	4,279461	0,283203	4,515382	0,277826	4,484745	0,259225	4,279461	0,289063	4,385705
	M3	0,246704	4,279463	0,233724	4,299067	0,227912	4,26787	0,25154	4,27946	0,233724	4,294394
	M4	0,266507	4,279458	0,191189	4,268377	0,19105	4,23837	0,270883	4,279458	0,203993	4,263204
	M5	0,253267	4,279465	0,144097	4,273181	0,140706	4,239486	0,260117	4,279459	0,287543	4,267675
	M6	0,208163	4,279461	0,14171	4,269591	0,138124	4,238416	0,257325	4,27946	0,289063	4,255151
	M7	0,219676	4,280461	0,242405	10,35623	0,250861	10,50695	0,243682	4,279481	0,348958	6,456837
	M8	0,231766	4,279472	0,229818	4,269593	0,235944	4,243399	0,270566	4,279462	0,290582	4,262372
	M9	0,227045	4,279459	0,273438	4,440142	0,281268	4,41858	0,28513	4,279458	0,290365	4,422265
	M10	0,240832	4,279462	0,258464	13,75288	0,268216	13,68374	0,261643	4,279459	0,292101	13,10581
	M11	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?

Ordered By Training		Last Iteration					
		Training		Validation		Testing	
		Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
O1_N	M11	0,980427	0,055931	0,336806	9535,638	0,345955	9355,894
	M	0,970813	0,086918	0,840278	0,942821	0,831325	0,92084
	M3	0,970641	0,090223	0,851345	0,826116	0,844664	0,820539
	M1	0,954723	0,123859	0,902127	0,353786	0,903184	0,358114
	M4	0,940562	0,156737	0,900608	0,30994	0,90132	0,311124
	M5	0,938259	0,158715	0,892144	0,299664	0,88253	0,345283
	M2	0,924558	0,193682	0,892795	0,310255	0,888985	0,316839
	M7	0,907806	0,221938	0,881293	0,320778	0,873064	0,340033
	M8	0,676703	0,683147	0,828993	0,437899	0,831038	0,457268
	M6	0,610443	0,829959	0,568793	0,899124	0,561388	0,912741
	M10	0,479362	1,094499	0,212891	1,810457	0,212134	1,823538
	M9	0,345864	1,333494	0,33941	1,341201	0,346816	1,336558
O1_A	M1	0,963358	0,096333	0,832899	138,9714	0,822576	139,6061
	M	0,954033	0,119799	0,855469	101,5868	0,852123	99,51452
	M3	0,94036	0,152931	0,820095	87,96473	0,817269	89,20181
	M4	0,928127	0,186115	0,89822	47,09663	0,893574	47,72771
	M2	0,918283	0,204798	0,840712	63,11361	0,836919	62,37116
	M5	0,759484	0,551318	0,538845	226,1006	0,527395	226,2507
	M6	0,705716	0,63784	0,710069	45,6938	0,718158	43,58591
	M7	0,67843	0,679675	0,699653	56,1796	0,682588	55,75502
	M8	0,569656	0,918822	0,494141	33,4912	0,497705	32,48192
	M9	0,563151	0,929742	0,262153	223,3725	0,258032	229,8507
	M10	0,458062	1,147688	0,190972	416,9665	0,193632	424,1612
	M11	0,2636	4,279483	0,142795	37,2626	0,137694	37,0009

Ordered By Validation		Last Iteration					
		Training		Validation		Testing	
		Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
O1_N	M1	0,954723	0,123859	0,902127	0,353786	0,903184	0,358114
	M4	0,940562	0,156737	0,900608	0,30994	0,90132	0,311124
	M2	0,924558	0,193682	0,892795	0,310255	0,888985	0,316839
	M5	0,938259	0,158715	0,892144	0,299664	0,88253	0,345283
	M7	0,907806	0,221938	0,881293	0,320778	0,873064	0,340033
	M3	0,970641	0,090223	0,851345	0,826116	0,844664	0,820539
	M	0,970813	0,086918	0,840278	0,942821	0,831325	0,92084
	M8	0,676703	0,683147	0,828993	0,437899	0,831038	0,457268
	M6	0,610443	0,829959	0,568793	0,899124	0,561388	0,912741
	M9	0,345864	1,333494	0,33941	1,341201	0,346816	1,336558
	M11	0,980427	0,055931	0,336806	9535,638	0,345955	9355,894
	M10	0,479362	1,094499	0,212891	1,810457	0,212134	1,823538
O1_A	M4	0,928127	0,186115	0,89822	47,09663	0,893574	47,72771
	M	0,954033	0,119799	0,855469	101,5868	0,852123	99,51452
	M2	0,918283	0,204798	0,840712	63,11361	0,836919	62,37116
	M1	0,963358	0,096333	0,832899	138,9714	0,822576	139,6061
	M3	0,94036	0,152931	0,820095	87,96473	0,817269	89,20181
	M6	0,705716	0,63784	0,710069	45,6938	0,718158	43,58591
	M7	0,67843	0,679675	0,699653	56,1796	0,682588	55,75502
	M5	0,759484	0,551318	0,538845	226,1006	0,527395	226,2507
	M8	0,569656	0,918822	0,494141	33,4912	0,497705	32,48192
	M9	0,563151	0,929742	0,262153	223,3725	0,258032	229,8507
	M10	0,458062	1,147688	0,190972	416,9665	0,193632	424,1612
	M11	0,2636	4,279483	0,142795	37,2626	0,137694	37,0009

ResNet Implementation

We also implemented a ResNet model to leverage its deep learning capabilities and residual connections for improved performance




```python
resnet50 = ResNet50(weights='imagenet',
                    input_shape=(128, 128, 3), # 3 channels because of imagenet
                    include_top=False)

def CustomNN11():
    x = GlobalAveragePooling2D()(resnet50.output)
    output = Dense(4, activation='softmax')(x)

    return Model(inputs=resnet50.input, outputs=output)
```

The significant discrepancy between the training and validation/testing performance metrics indicates that the model is overfitting. Overfitting occurs when a model learns the training data too well.

	Training		Validation		Testing	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
M11	0,980427	0,055931	0,336806	9535,638	0,345955	9355,894

Optimizers

We evaluated several optimizers to find the best one for our best 7 models. The optimizers considered were :

Adam, Adadelta, Adafactor, AdamW, Lion, Nadam, RMSprop, and SGD.

Optimizers are algorithms or methods used to **minimize** an error function (*loss function*).

Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.

During the test simulations, **10 epochs** turned out to be the ideal number during hypertuning.

Accuracy - Last Iteration

	Training							Validation							Testing						
	M	M1	M2	M3	M4	M5	M7	M	M1	M2	M3	M4	M5	M7	M	M1	M2	M3	M4	M5	M7
Adam	0,954033	0,963358	0,918283	0,94036	0,928127	0,759484	0,67843	0,855469	0,832899	0,840712	0,820095	0,89822	0,538845	0,699653	0,852123	0,822576	0,836919	0,817269	0,893574	0,527395	0,682588
Adadelta	0,374964	0,356456	0,345864	0,420557	0,345864	0,345864	0,401387	0,359809	0,348307	0,335938	0,395399	0,339844	0,339193	0,161241	0,368617	0,351692	0,345525	0,409495	0,345668	0,345955	0,153901
Adafactor	0,228254	0,228254	0,228254	0,228254	0,228254	0,228254	0,562835	0,232856	0,229384	0,232856	0,229384	0,231554	0,232856	0,215278	0,235657	0,234796	0,235657	0,234796	0,236948	0,235657	0,206397
AdamW	0,822952	0,892867	0,798544	0,813367	0,840395	0,693167	0,447527	0,556207	0,738064	0,634549	0,643446	0,698568	0,527778	0,424045	0,55938	0,743115	0,640419	0,648164	0,707831	0,513626	0,423551
Lion	0,763399	0,85801	0,733636	0,912527	0,941397	0,838725	0,477808	0,610243	0,648003	0,669922	0,529948	0,825304	0,524957	0,290582	0,615462	0,662077	0,667384	0,538153	0,826592	0,524383	0,28012
Nadam	0,847648	0,924299	0,825082	0,826694	0,879685	0,706954	0,518047	0,553602	0,715929	0,712674	0,689453	0,873481	0,560764	0,401259	0,561388	0,722605	0,701807	0,691767	0,873637	0,553787	0,413655
RMSprop	0,94937	0,966669	0,916153	0,791578	0,847274	0,733723	0,534914	0,730035	0,895616	0,781033	0,532552	0,547743	0,42296	0,495009	0,729633	0,88253	0,767499	0,526965	0,557803	[illegible]	[illegible]
SGD	0,861004	0,619452	0,585257	0,594324	0,650596	0,437309	0,580421	0,847656	0,618707	0,617839	0,557075	0,667969	0,345486	0,547526	0,836489	0,62292	0,619191	[illegible]	0,678571	0,348967	0,54848

Loss - Last Iteration

	Training							Validation							Testing						
	M	M1	M2	M3	M4	M5	M7	M	M1	M2	M3	M4	M5	M7	M	M1	M2	M3	M4	M5	M7
Adam	0,119799	0,096333	0,204798	0,152931	0,186115	0,551318	0,679675	101,5868	138,9714	63,11361	87,96473	47,09663	226,1006	56,1796	99,51452	139,6061	62,37116	89,20181	47,72771	226,2507	55,75502
Adadelta	1,278969	1,299899	1,327074	1,212545	1,33035	1,328425	1,260432	69,46381	72,05654	73,7252	73,69516	65,339	72,66269	307,0273	68,00081	71,11269	73,03715	70,54932	64,7852	72,09054	312,3015
Adafactor	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	0,947572	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	463,9055	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	472,7566
AdamW	0,420816	0,26456	0,466583	0,43649	0,367502	0,665612	1,127606	155,0865	122,7811	109,7304	95,35262	117,8444	120,72	238,5759	149,7148	115,1331	109,3709	94,55078	111,6366	121,3821	238,6244
Lion	0,541392	0,338286	0,588261	0,221017	0,152744	0,385403	1,071359	156,2725	298,7594	114,6819	459,5096	98,99802	449,0855	288626,6	156,1677	285,8157	110,332	456,7946	92,24287	441,8654	292101,4
Nadam	0,367388	0,198644	0,410563	0,409347	0,295626	0,633756	1,001187	361,4565	349,0181	109,1732	109,8621	41,76082	123,3504	679,751	357,5465	329,2043	114,5991	106,0737	41,81086	122,4244	688,7033
RMSprop	0,137531	0,096035	0,220925	0,490982	0,371667	0,599858	0,97431	242,0475	58,54897	141,8343	403,0045	160,4866	479,3727	305,3133	255,1473	62,51388	146,7948	403,9467	154,1485	471,9396	317,6852
SGD	0,33984	0,808954	0,879242	0,870305	0,74494	1,187497	0,880449	35,7982	79,12017	59,53105	98,57728	57,82984	194,1437	82,22006	37,81538	75,40391	59,0852	95,10289	56,73981	195,9379	82,08315

Accuracy - Max

	Training							Validation							Testing						
	M	M1	M2	M3	M4	M5	M7	M	M1	M2	M3	M4	M5	M7	M	M1	M2	M3	M4	M5	M7
Adam	0,954033	0,963358	0,918283	0,94036	0,928127	0,759484	0,67843	0,894531	0,898438	0,894314	0,825955	0,905816	0,538845	0,72526	0,852123	0,822576	0,836919	0,817269	0,893574	0,527395	0,682588
Adadelta	0,374964	0,356456	0,345864	0,420557	0,345864	0,345864	0,401387	0,359809	0,348307	0,339193	0,395399	0,339844	0,339193	0,355686	0,368617	0,351692	0,345525	0,409495	0,345668	0,345955	0,153901
Adafactor	0,228254	0,22834	0,228254	0,228455	0,22834	0,228398	0,562835	0,232856	0,231771	0,232856	0,231771	0,232856	0,232856	0,505208	0,235657	0,234796	0,235657	0,234796	0,236948	0,235657	0,206397
AdamW	0,822952	0,892867	0,798544	0,813367	0,840395	0,693167	0,447527	0,678602	0,738064	0,651693	0,643446	0,742405	0,527778	0,440538	0,55938	0,743115	0,640419	0,648164	0,707831	0,513626	0,423551
Lion	0,763399	0,85801	0,733636	0,912527	0,941397	0,838725	0,477808	0,665799	0,661024	0,69184	0,614149	0,852214	0,570964	0,469618	0,615462	0,662077	0,667384	0,538153	0,826592	0,524383	0,28012
Nadam	0,847648	0,924299	0,825082	0,826694	0,879685	0,706954	0,518047	0,631944	0,715929	0,722656	0,689453	0,873481	0,560764	0,492622	0,561388	0,722605	0,701807	0,691767	0,873637	[illegible]	[illegible]
RMSprop	0,94937	0,966669	0,916153	0,791578	0,847274	0,733723	0,534914	0,837457	0,895616	0,848307	0,614149	0,708333	0,470486	0,495009	0,729633	0,88253	0,767499	[illegible]	0,557803	0,428571	0,498279
SGD	0,861004	0,619452	0,585257	0,594324	0,650596	0,437309	0,580421	0,869575	0,618707	0,617839	0,557075	0,667969	0,350911	0,559245	0,836489	0,62292	[illegible]	0,571859	0,678571	0,348967	0,54848

Loss - Min

	Training							Validation							Testing						
	M	M1	M2	M3	M4	M5	M7	M	M1	M2	M3	M4	M5	M7	M	M1	M2	M3	M4	M5	M7
Adam	0,119799	0,096333	0,204798	0,152931	0,186115	0,551318	0,679675	52,80975	46,87912	30,77151	74,01408	34,50066	73,03018	13,27193	99,51452	139,6061	62,37116	89,20181	47,72771	226,2507	55,75502
Adadelta	1,278969	1,299899	1,327074	1,212545	1,33035	1,328425	1,260432	69,46381	68,25607	44,92303	70,49592	9,15693	21,40174	133,7904	68,00081	71,11269	73,03715	70,54932	64,7852	72,09054	312,3015
Adafactor	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	0,947572	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	116,1893	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	#NOME?	472,7566
AdamW	0,420816	0,26456	0,466583	0,43649	0,367502	0,665612	1,127606	62,82343	58,79314	60,86594	86,89916	48,3206	93,68549	89,24568	149,7148	115,1331	109,3709	94,55078	111,6366	121,3821	238,6244
Lion	0,541392	0,338286	0,588261	0,221017	0,152744	0,385403	1,071359	82,68707	73,52971	60,92074	120,0454	60,72481	228,3328	64,28588	156,1677	285,8157	110,332	456,7946	92,24287	441,8654	292101,4
Nadam	0,367388	0,198644	0,410563	0,409347	0,295626	0,633756	1,001187	193,5978	200,8936	84,74699	82,56892	32,55912	78,33354	29,92693	357,5465	329,2043	114,5991	106,0737	41,81086	122,4244	688,7033
RMSprop	0,137531	0,096035	0,220925	0,490982	0,371667	0,599858	0,97431	59,2989	40,99645	66,39182	90,32619	49,75314	80,94553	267,4594	255,1473	62,51388	146,7948	403,9467	154,1485	471,9396	317,6852
SGD	0,33984	0,808954	0,879242	0,870305	0,74494	1,187497	0,880449	28,36656	76,96352	58,49344	98,57728	13,63396	104,5695	74,13764	37,81538	75,40391	59,0852	95,10289	56,73981	195,9379	82,08315

Accuracy - Last Iteration: Accuracy metrics at the final iteration of training.

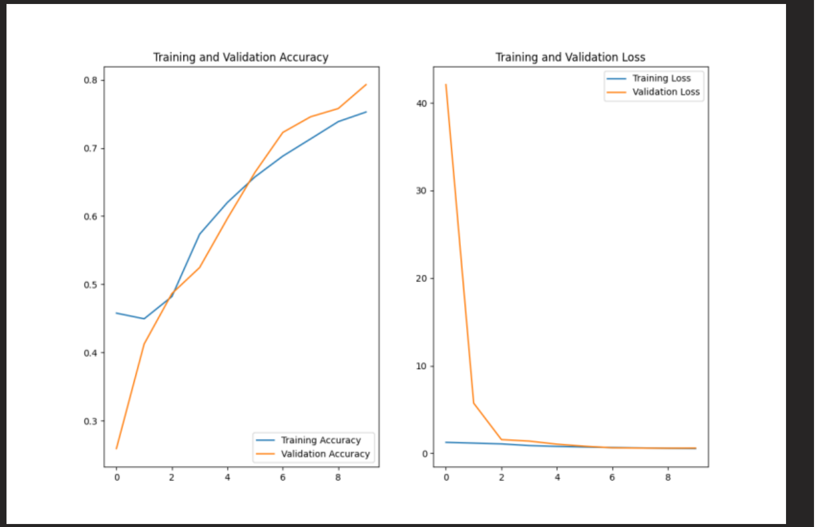
Loss - Last Iteration: Lower loss values indicate better model performance and fewer errors.

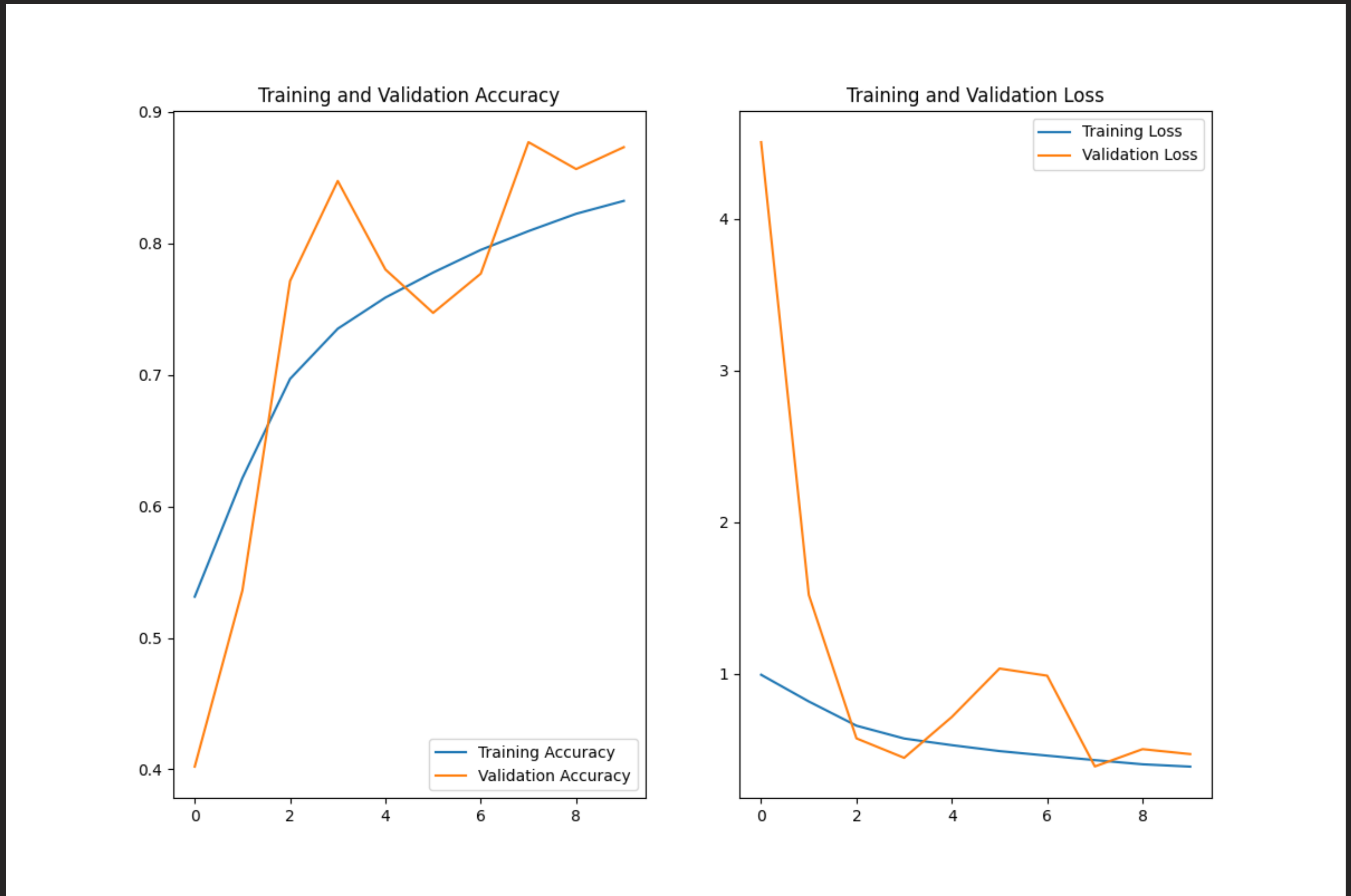
Accuracy - Max: Maximum accuracy achieved.

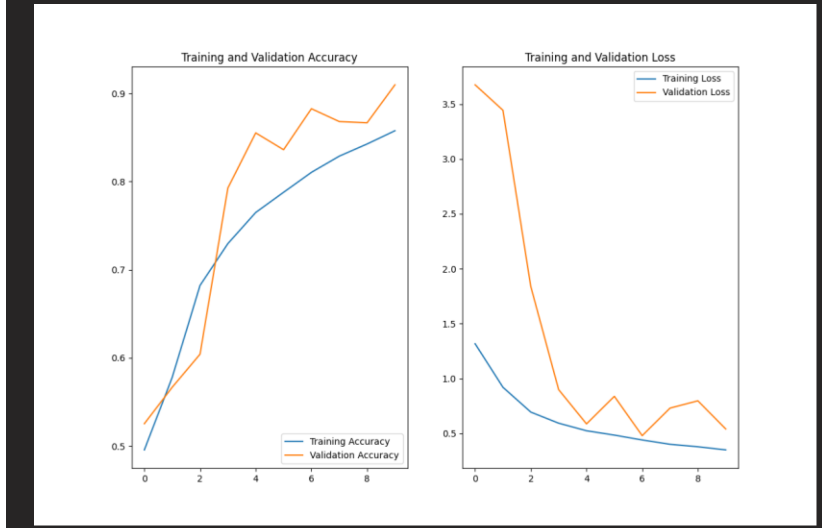
Loss - Min: Minimum loss values achieved.

Model's performances

The following graphs show training and validation accuracy/loss for three different models across epochs. As demonstrated, the models did not exhibit signs of overfitting or underfitting, indicating a great generalization capabilities on the test data.







Ensemble Neural Networks with Majority Voting

Purpose of Ensemble Neural Networks

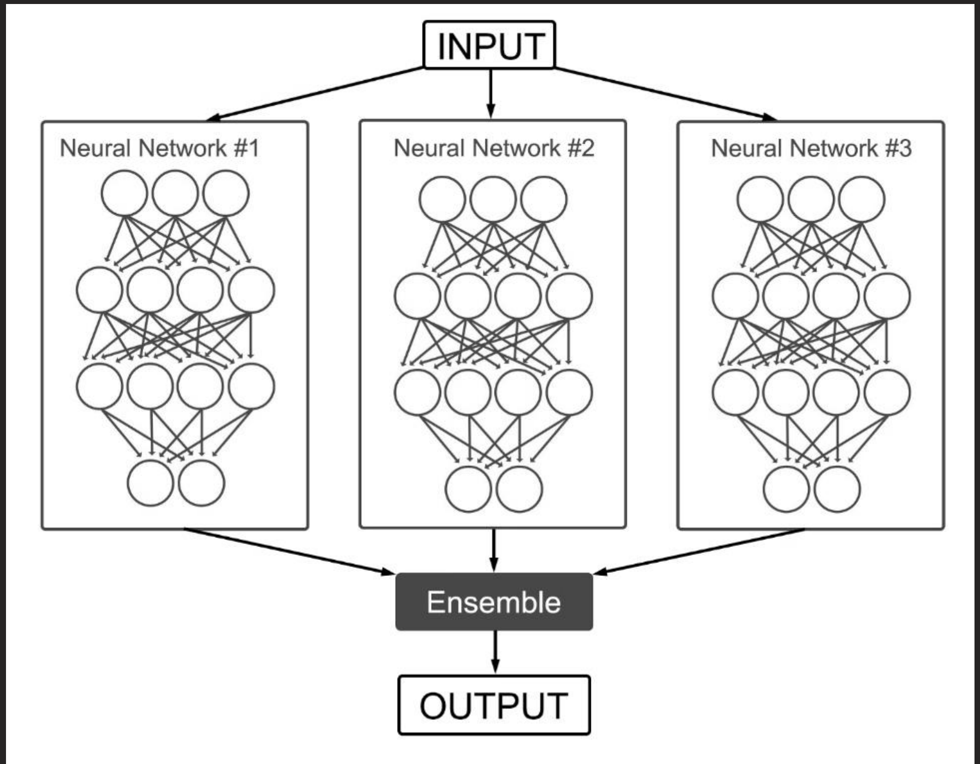
1. An ensemble neural network combines the predictions of multiple models to improve overall performance.

Advantages:

1. **Improved Accuracy:** Enhances accuracy by combining outputs of different models.
2. **Increased Robustness:** Reduces variance and overfitting risk by leveraging diverse model predictions.
3. **Better Generalization:** Improves generalization to new, unseen data.

Implementation in Our Project

1. We used the **Majority Voting** technique (both weighted and unweighted).
2. Implemented two types of ensembles: **Ensemble of 3 Networks, Ensemble of 5 Networks**




```
[ ] # Majority Vote Model (re-implemented v2)
    class CustomNN12tris(tf.keras.Model):
        def __init__(self, models, accuracies):
            super(CustomNN12tris, self).__init__()
            self.models = models
            self.accuracies = accuracies

        def call(self, inputs):
            predictions = tf.convert_to_tensor([model(inputs) for model in self.models])

            # Get votes of each model (weighted based on the accuracy of the model)
            #votes = np.sum(predictions * self.accuracies, axis=0)
            for index, model_votes in enumerate(predictions):
                model_votes *= self.accuracies[index]
            votes = np.sum(predictions, axis=0)


            # Normalize the votes
            return tf.convert_to_tensor(votes / votes.sum(axis=1)[:, np.newaxis])
```

Results of Ensemble Model

The ensemble models showed a significant improvement in accuracy. For example, using 5 models with NoSplit and Duplication (NoSplit&Dup), NN12tris achieved an accuracy of **92.67%**

NN12bis

- Basic majority voting.

NN12tris

- Weighted majority voting on each validation accuracy.

Accuracy

3 Models

	NoSplit	NoSplit&Dup	Split	Split&Dup V1	Split&Dup V2
NN12bis	0,879999995	0,912068963	0,785172403	0,408735633	0,703793108
NN12tris	0,881264389	0,9265517	0,787011504	0,713793099	0,755862057

5 Models

	NoSplit	NoSplit&Dup	Split	Split&Dup V1	Split&Dup V2
NN12bis	0,911954045	0,918965518	0,794942558	0,71850574	0,790689647
NN12tris	0,912758648	0,926666677	0,798505723	0,831724167	0,847356319

Loss

3 Models

	NoSplit	NoSplit&Dup	Split	Split&Dup V1	Split&Dup V2
NN12bis	0,620373845	0,487590641	1,298410296	2,016934156	1,394545436
NN12tris	0,603228152	0,238315463	1,270070672	0,834860682	0,61325264

5 Models

	NoSplit	NoSplit&Dup	Split	Split&Dup V1	Split&Dup V2
NN12bis	0,359499216	0,393530726	0,848718107	0,967556596	0,806558788
NN12tris	0,346989453	0,250947058	0,825519383	0,65546453	0,605145037

NoSplit : All models trained on the same augmented dataset.

NoSplit&Dup : Like NoSplit, but with the addition of the original non-augmented dataset.

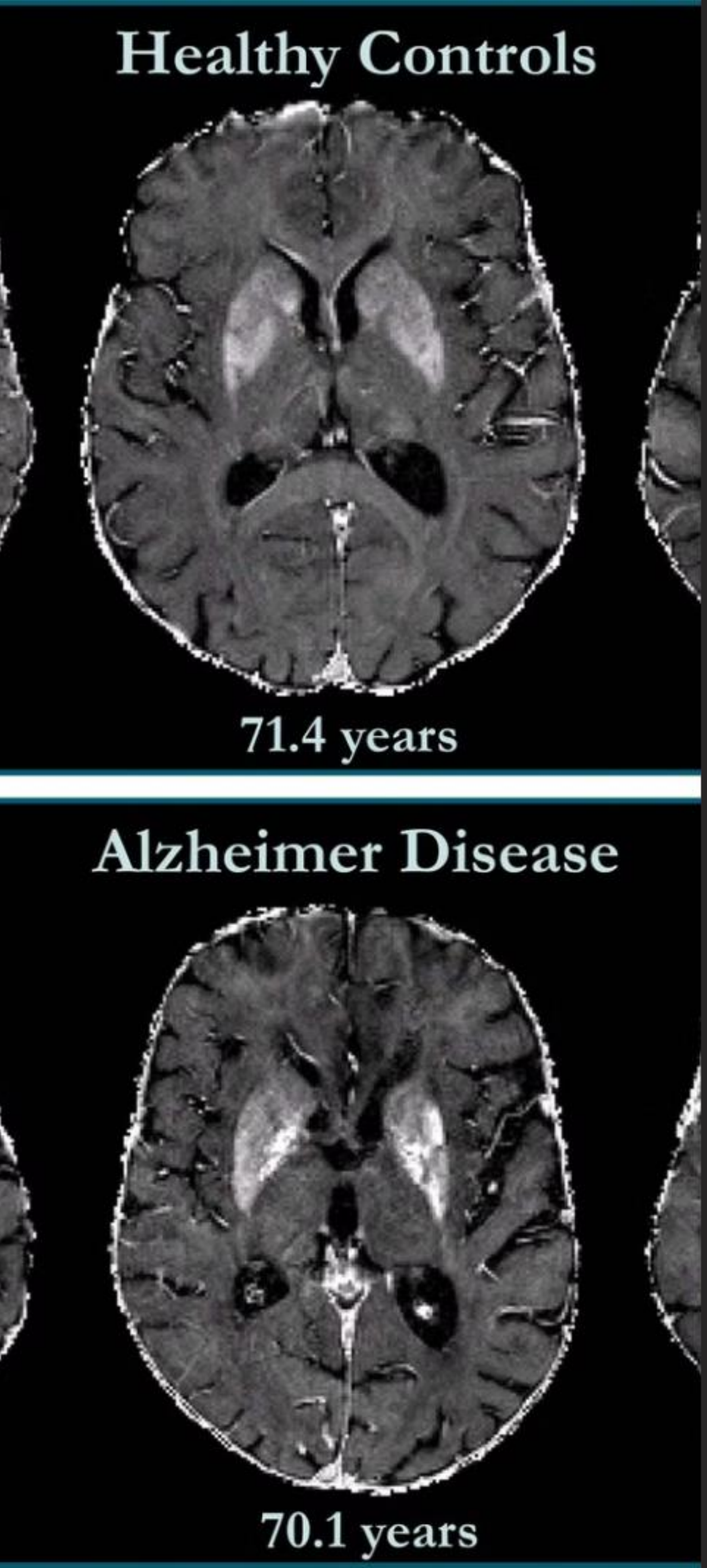
Split : Each model receives an independent portion of the dataset, with augmentation.

Split&Dup

Each model receives an independent portion of the augmented dataset, + original dataset.

- **V1**: Base dataset + augmented (mode 2, mode 0, and augmented of mode 2).

- **V2**: Base dataset + augmented.



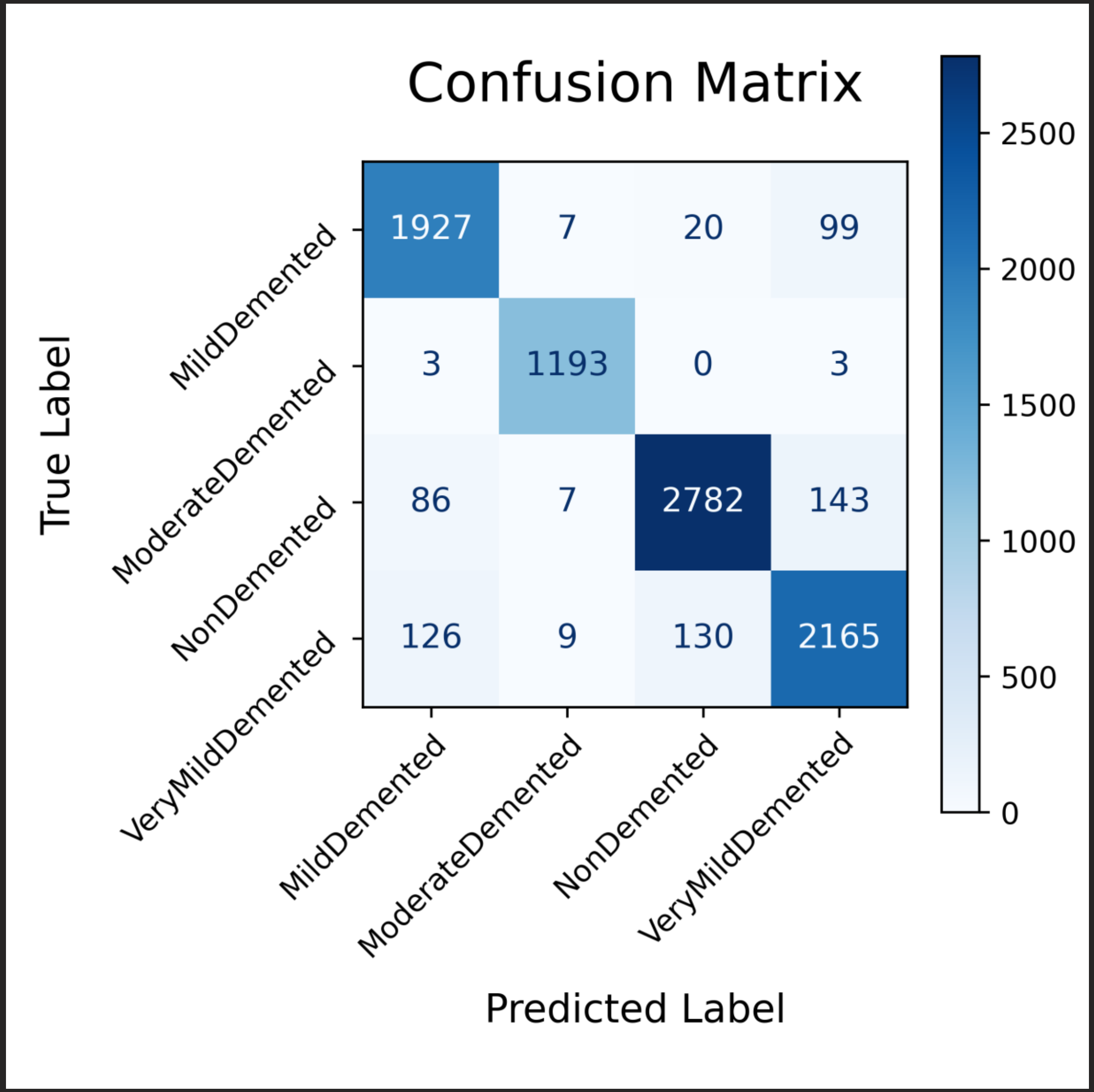
Results Obtained and Conclusion

1 Summary

This project successfully developed a deep learning learning model capable of classifying MRI images images into four distinct stages of Alzheimer's Alzheimer's disease with accuracy of **92.67%**. **92.67%**.

2 Confusion Matrix

The confusion matrix is a tool used to evaluate the performance of our classification model. It displays the model's predictions against the actual outcomes. Here's the confusion matrix for our model:



3 Impact Reflection

This research has the potential to significantly impact clinical practices by enabling more accurate and earlier detection of Alzheimer's disease.

References

List all the references used in the project, including scientific papers, technical documents, and other relevant sources.

Articles and Studies:

1. **Title:** Alzheimer's disease detection from structural MRI using sparse coding and linear regression. **Link:** ScienceDirect
2. **Title:** Evaluation of Neuro Images for the Detection of Alzheimer's Disease. **Link:** Frontiers in Public Health
3. **Title:** The effect of data augmentation and 3D-CNN deep learning model in Alzheimer's disease detection. **Link:** arXiv

Documentation and Tools:

1. **Title:** How to choose loss functions when training deep learning neural networks **Link:** Machine Learning Mastery
2. **Title:** Documentation on Kullback-Leibler Divergence Loss in PyTorch **Link:** PyTorch KLDivLoss
3. **Title:** Documentation on loss functions in PyTorch **Link:** PyTorch Loss Functions
4. **Title:** Harvard CS50's Artificial Intelligence with Python **Link:** Harvard CS50's AI with Python

Thank you for attention!