Enhancing Hair Classification through Hyperparameter Tuning in a CNN Model

Cyrus Benjamin C. Canape
Edrine Frances A. Esguerra
[#]*Department of Computer Science, University of Santo Tomas-Manila*
España Bldv., Sampaloc, Manila, 1008 Metro Manila, Philippines
[1]`cyrusbenjamin.canape.cics@ust.edu.ph`
[2]`edrinefrances.esguerra.cics@ust.edu.ph`

***Abstract*— This paper presents a comprehensive analysis of convolutional neural network (CNN) hyperparameters on hair type classification. Eight hyperparameters were modified to evaluate their impact on model performance: number of layers, filters, kernel size, dilation rate, learning rate, dropout rate, batch normalization, and max pooling. Results indicate that batch normalization provided the most significant improvement in classification accuracy, followed by max pooling and dilation rate adjustments. The study reveals important insights into CNN optimization for image classification tasks, particularly for distinguishing between curly, straight, and wavy hair types.**

***Keywords*— Convolutional Neural Networks, Hyperparameter Tuning, Image Classification, Computer Vision, Hair Classification**

I. Introduction

Image classification using convolutional neural networks (CNNs) has become a mainstream approach for various computer vision tasks. The effectiveness of these networks heavily depends on their architecture and hyperparameter configuration (Bergstra, 2012). This paper focuses on classifying hair types into three categories: curly, straight, and wavy, and investigates the effects of eight different hyperparameters on model performance: number of layers, number of filters, kernel size, dilation rate, learning rate, dropout rate, batch normalization, and max pooling.

The primary objective is to understand how specific architectural and training parameters influence the classification accuracy and convergence behavior of CNN models. By analyzing the impact of each hyperparameter independently, we aim to identify optimal configurations for hair classification and derive insights that may extend to other image classification tasks.

The dataset consists of images categorized into three hair types: Curly, Straight, and Wavy. It consists of 332 entries with curly hair, 318 with straight hair, and 331 with wavy hair. All images were organized in directories where folder names served as class labels.

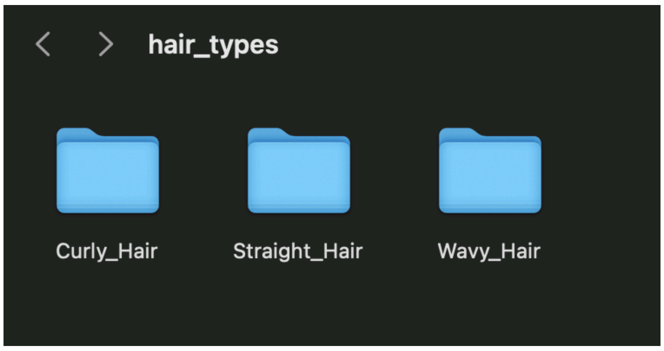


Fig. 1 Dataset

The following sections detail our methodology for data preparation, model architecture, and experimental design used to evaluate hyperparameter effects on classification performance.

II. Methodology

This study employs convolutional neural networks for hair classification, building upon foundational work in deep learning. The methodology follows established practices in CNN architecture design while investigating the effects of various hyperparameters.

A. *Hyperparameters*

This study investigates eight key hyperparameters to understand their impact on CNN performance for hair classification:

1. **Number of Layers:** Controls the depth of the network, with deeper networks potentially capturing more complex hierarchical features.

2. **Number of Filters:** Determines feature detection capacity at each layer, with more filters enabling detection of more diverse patterns.

3. **Kernel Size**: Defines the spatial extent of convolutional operations, affecting the scale of detectable features.

4. **Dilation Rate:** Expands the receptive field without increasing parameters by introducing gaps between kernel elements.

5. **Learning Rate:** Controls step size during optimization, affecting convergence speed and stability.

6. **Dropout Rate:** Implements regularization by randomly deactivating neurons during training to prevent co-adaptation.

7. **Batch Normalization:** Normalizes layer inputs to reduce internal covariate shift, stabilizing and accelerating training.

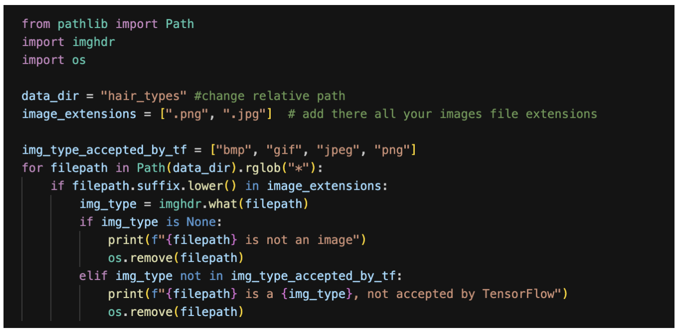
8. **Max Pooling:** Reduces spatial dimensions while preserving dominant features, providing translation invariance.

Each hyperparameter was varied independently while keeping others constant to isolate their specific effects on model performance.

B. *Data Preparation*

The dataset was loaded using TensorFlow's image_dataset_from_directory utility. The dataset includes only JPEG and PNG formats to ensure compatibility with the TensorFlow processing framework. Image preprocessing included:

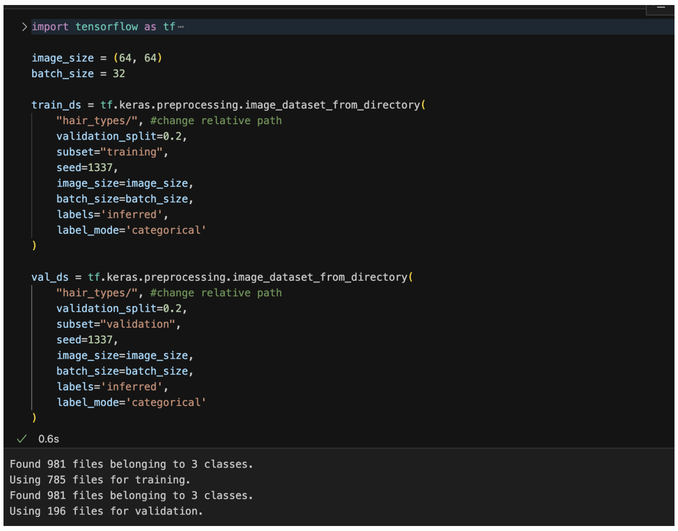
1. Resizing to 64×64 pixels
2. Filtering unsupported image types



```python
from pathlib import Path
import imghdr
import os

data_dir = "hair_types" #change relative path
image_extensions = [".png", ".jpg"]  # add there all your images file extensions

img_type_accepted_by_tf = ["bmp", "gif", "jpeg", "png"]
for filepath in Path(data_dir).rglob("*"):
    if filepath.suffix.lower() in image_extensions:
        img_type = imghdr.what(filepath)
        if img_type is None:
            print(f"{filepath} is not an image")
            os.remove(filepath)
        elif img_type not in img_type_accepted_by_tf:
            print(f"{filepath} is a {img_type}, not accepted by TensorFlow")
            os.remove(filepath)
```

Fig. 2 Image Filtering

3. Categorical encoding for multi-class classification
4. Batch creation (32 images per batch)



```python
> import tensorflow as tf…

  image_size = (64, 64)
  batch_size = 32

  train_ds = tf.keras.preprocessing.image_dataset_from_directory(
      "hair_types/", #change relative path
      validation_split=0.2,
      subset="training",
      seed=1337,
      image_size=image_size,
      batch_size=batch_size,
      labels='inferred',
      label_mode='categorical'
  )

  val_ds = tf.keras.preprocessing.image_dataset_from_directory(
      "hair_types/", #change relative path
      validation_split=0.2,
      subset="validation",
      seed=1337,
      image_size=image_size,
      batch_size=batch_size,
      labels='inferred',
      label_mode='categorical'
  )
✓ 0.6s
```

```
Found 981 files belonging to 3 classes.
Using 785 files for training.
Found 981 files belonging to 3 classes.
Using 196 files for validation.
```

Fig. 2 Dataset Preprocessing

C. Base Model Architecture

The baseline CNN model follows a sequential architecture with the following structure:

1. **Input Layer:**
   - Accepts images with dimensions 64×64×3
   - Rescales pixel values from [0-255] to [0-1] range

2. **Convolutional Blocks:**
   - First Block: 4 filters with 16×16 kernel size, stride of 1, 'valid' padding, dilation rate of 1, ReLU activation

- Second Block: 8 filters with 8×8 kernel size, stride of 1, 'valid' padding, dilation rate of 1, ReLU activation
- Third Block: 16 filters with 4×4 kernel size, stride of 1, 'valid' padding, dilation rate of 1, ReLU activation

3. **Output Layers:**
   - Global Average Pooling to reduce spatial dimensions
   - Dense layer with 64 units and ReLU activation
   - Output dense layer with 3 units (one for each hair type class)
   - Softmax activation for multi-class probability output

4. **Compilation**:
   - Optimizer: Adam with default learning rate (0.001)
   - Loss function: Categorical cross-entropy
   - Evaluation metric: Accuracy

D. Evaluation Metrics

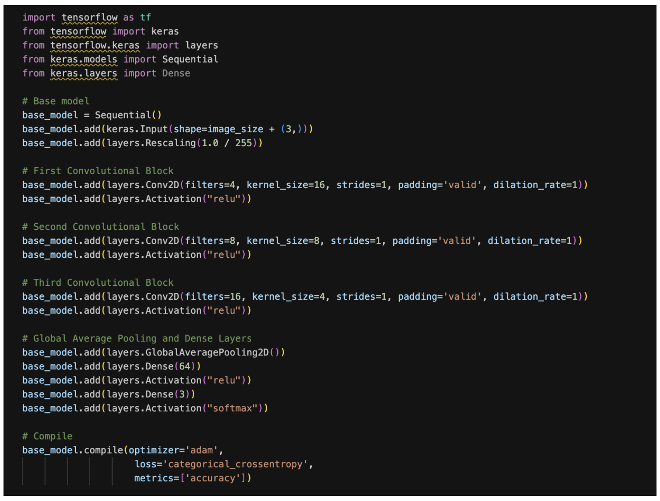
To evaluate model performance, four key metrics were tracked throughout training:

1. **Training Accuracy:** The proportion of correctly classified samples in the training set, measuring the model's ability to learn from the data.

2. **Training Loss:** The categorical cross-entropy loss on the training set, quantifying the difference between predicted and actual class distributions.

3. **Validation Accuracy:** The proportion of correctly classified samples in the validation set, assessing the model's generalization capability.

4. **Validation Loss:** The categorical cross-entropy loss on the validation set, providing insight into potential overfitting when compared with training loss.



```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from keras.models import Sequential
from keras.layers import Dense

# Base model
base_model = Sequential()
base_model.add(keras.Input(shape=image_size + (3,)))
base_model.add(layers.Rescaling(1.0 / 255))

# First Convolutional Block
base_model.add(layers.Conv2D(filters=4, kernel_size=16, strides=1, padding='valid', dilation_rate=1))
base_model.add(layers.Activation("relu"))

# Second Convolutional Block
base_model.add(layers.Conv2D(filters=8, kernel_size=8, strides=1, padding='valid', dilation_rate=1))
base_model.add(layers.Activation("relu"))

# Third Convolutional Block
base_model.add(layers.Conv2D(filters=16, kernel_size=4, strides=1, padding='valid', dilation_rate=1))
base_model.add(layers.Activation("relu"))

# Global Average Pooling and Dense Layers
base_model.add(layers.GlobalAveragePooling2D())
base_model.add(layers.Dense(64))
base_model.add(layers.Activation("relu"))
base_model.add(layers.Dense(3))
base_model.add(layers.Activation("softmax"))

# Compile
base_model.compile(optimizer='adam',
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])
```

Fig. 3 Base Model

III. Experiments

The experiments in this study were designed to evaluate the impact of individual hyperparameters on hair classification performance.

Each experiment modified a single hyperparameter while keeping all others constant, creating a controlled environment to isolate the specific effects of each modification. All models were trained and evaluated using identical datasets, preprocessing steps, and evaluation metrics to ensure fair comparison. This methodical approach allows for direct attribution of performance differences to the specific hyperparameter being investigated.

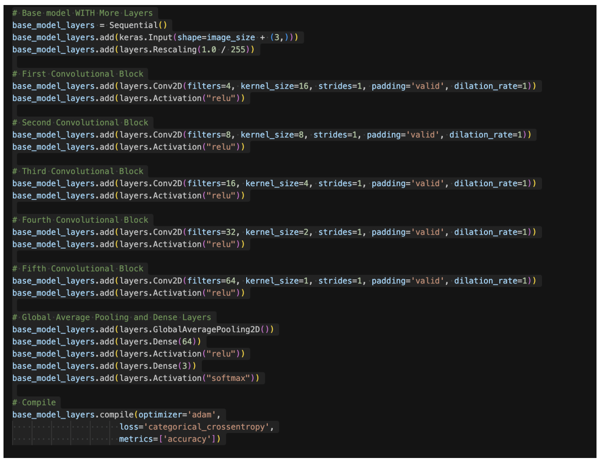
This experiment tests whether deeper networks can capture more complex hierarchical features in hair textures.

A. Hyperparameter Variations

Eight model variations were developed, each modifying a single hyperparameter while keeping others constant:

1. **Increased Number of Layers:** Added two additional convolutional blocks (five total)

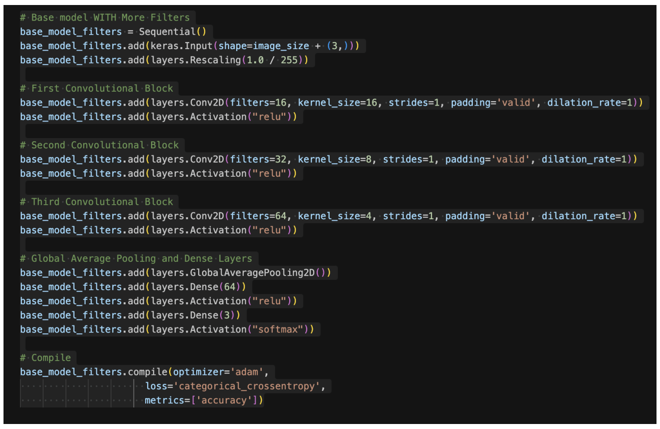
- Fourth Block: 32 filters with 2×2 kernel size, ReLU activation
- Fifth Block: 64 filters with 1×1 kernel size, ReLU activation



```python
# Base model WITH More Layers
base_model_layers = Sequential()
base_model_layers.add(keras.Input(shape=image_size + (3,)))
base_model_layers.add(layers.Rescaling(1.0 / 255))

# First Convolutional Block
base_model_layers.add(layers.Conv2D(filters=4, kernel_size=16, strides=1, padding='valid', dilation_rate=1))
base_model_layers.add(layers.Activation("relu"))

# Second Convolutional Block
base_model_layers.add(layers.Conv2D(filters=8, kernel_size=8, strides=1, padding='valid', dilation_rate=1))
base_model_layers.add(layers.Activation("relu"))

# Third Convolutional Block
base_model_layers.add(layers.Conv2D(filters=16, kernel_size=4, strides=1, padding='valid', dilation_rate=1))
base_model_layers.add(layers.Activation("relu"))

# Fourth Convolutional Block
base_model_layers.add(layers.Conv2D(filters=32, kernel_size=2, strides=1, padding='valid', dilation_rate=1))
base_model_layers.add(layers.Activation("relu"))

# Fifth Convolutional Block
base_model_layers.add(layers.Conv2D(filters=64, kernel_size=1, strides=1, padding='valid', dilation_rate=1))
base_model_layers.add(layers.Activation("relu"))

# Global Average Pooling and Dense Layers
base_model_layers.add(layers.GlobalAveragePooling2D())
base_model_layers.add(layers.Dense(64))
base_model_layers.add(layers.Activation("relu"))
base_model_layers.add(layers.Dense(3))
base_model_layers.add(layers.Activation("softmax"))

# Compile
base_model_layers.compile(optimizer='adam',
                          loss='categorical_crossentropy',
                          metrics=['accuracy'])
```

Fig. 4 Experiment 1 - Base Model with More Layers

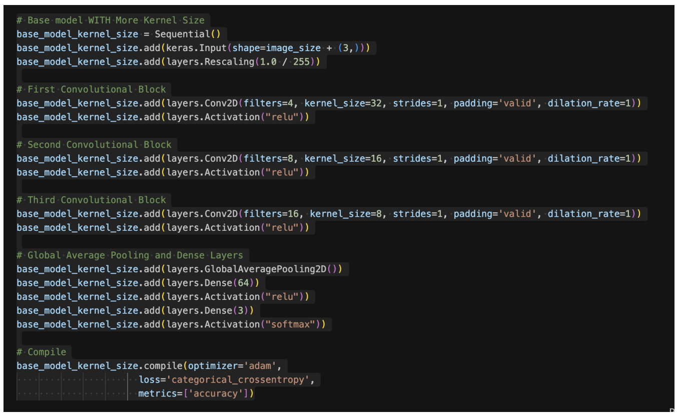
2. **Increased Number of Filters:** Increased filters in each block
- First block: 16 filters (instead of 4)
- Second block: 32 filters (instead of 8)
- Third block: 64 filters (instead of 16)



```python
# Base model WITH More Filters
base_model_filters = Sequential()
base_model_filters.add(keras.Input(shape=image_size + (3,)))
base_model_filters.add(layers.Rescaling(1.0 / 255))

# First Convolutional Block
base_model_filters.add(layers.Conv2D(filters=16, kernel_size=16, strides=1, padding='valid', dilation_rate=1))
base_model_filters.add(layers.Activation("relu"))

# Second Convolutional Block
base_model_filters.add(layers.Conv2D(filters=32, kernel_size=8, strides=1, padding='valid', dilation_rate=1))
base_model_filters.add(layers.Activation("relu"))

# Third Convolutional Block
base_model_filters.add(layers.Conv2D(filters=64, kernel_size=4, strides=1, padding='valid', dilation_rate=1))
base_model_filters.add(layers.Activation("relu"))

# Global Average Pooling and Dense Layers
base_model_filters.add(layers.GlobalAveragePooling2D())
base_model_filters.add(layers.Dense(64))
base_model_filters.add(layers.Activation("relu"))
base_model_filters.add(layers.Dense(3))
base_model_filters.add(layers.Activation("softmax"))

# Compile
base_model_filters.compile(optimizer='adam',
                           loss='categorical_crossentropy',
                           metrics=['accuracy'])
```

Fig. 5 Experiment 2 - Base Model with More Filter

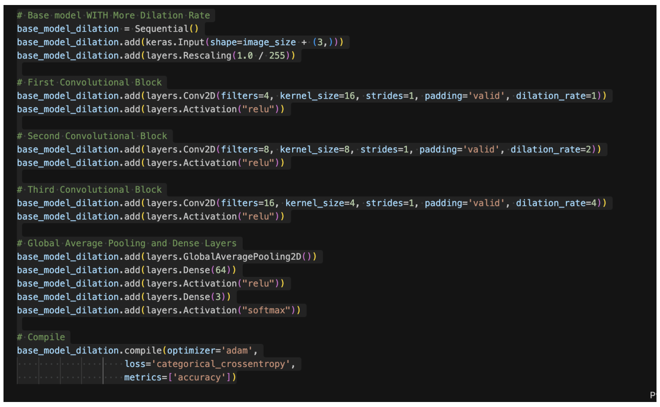
3. **Increased Kernel Size:** Enlarged kernel dimensions
- First block: 32×32 kernel (instead of 16×16)
- Second block: 16×16 kernel (instead of 8×8)
- Third block: 8×8 kernel (instead of 4×4)



```python
# Base model WITH More Kernel Size
base_model_kernel_size = Sequential()
base_model_kernel_size.add(keras.Input(shape=image_size + (3,)))
base_model_kernel_size.add(layers.Rescaling(1.0 / 255))

# First Convolutional Block
base_model_kernel_size.add(layers.Conv2D(filters=4, kernel_size=32, strides=1, padding='valid', dilation_rate=1))
base_model_kernel_size.add(layers.Activation("relu"))

# Second Convolutional Block
base_model_kernel_size.add(layers.Conv2D(filters=8, kernel_size=16, strides=1, padding='valid', dilation_rate=1))
base_model_kernel_size.add(layers.Activation("relu"))

# Third Convolutional Block
base_model_kernel_size.add(layers.Conv2D(filters=16, kernel_size=8, strides=1, padding='valid', dilation_rate=1))
base_model_kernel_size.add(layers.Activation("relu"))

# Global Average Pooling and Dense Layers
base_model_kernel_size.add(layers.GlobalAveragePooling2D())
base_model_kernel_size.add(layers.Dense(64))
base_model_kernel_size.add(layers.Activation("relu"))
base_model_kernel_size.add(layers.Dense(3))
base_model_kernel_size.add(layers.Activation("softmax"))

# Compile
base_model_kernel_size.compile(optimizer='adam',
                               loss='categorical_crossentropy',
                               metrics=['accuracy'])
```

Fig. 6 Experiment 3 - Base Model with More Kernel Size

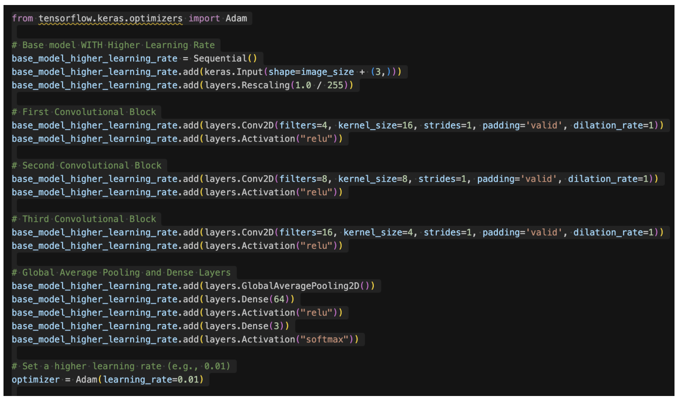
4. **Modified Dilation Rate:** Increased dilation progressively
- First block: dilation rate of 1
- Second block: dilation rate of 2
- Third block: dilation rate of 4



```python
# Base model WITH More Dilation Rate
base_model_dilation = Sequential()
base_model_dilation.add(keras.Input(shape=image_size + (3,)))
base_model_dilation.add(layers.Rescaling(1.0 / 255))

# First Convolutional Block
base_model_dilation.add(layers.Conv2D(filters=4, kernel_size=16, strides=1, padding='valid', dilation_rate=1))
base_model_dilation.add(layers.Activation("relu"))

# Second Convolutional Block
base_model_dilation.add(layers.Conv2D(filters=8, kernel_size=8, strides=1, padding='valid', dilation_rate=2))
base_model_dilation.add(layers.Activation("relu"))

# Third Convolutional Block
base_model_dilation.add(layers.Conv2D(filters=16, kernel_size=4, strides=1, padding='valid', dilation_rate=4))
base_model_dilation.add(layers.Activation("relu"))

# Global Average Pooling and Dense Layers
base_model_dilation.add(layers.GlobalAveragePooling2D())
base_model_dilation.add(layers.Dense(64))
base_model_dilation.add(layers.Activation("relu"))
base_model_dilation.add(layers.Dense(3))
base_model_dilation.add(layers.Activation("softmax"))

# Compile
base_model_dilation.compile(optimizer='adam',
                            loss='categorical_crossentropy',
                            metrics=['accuracy'])
```

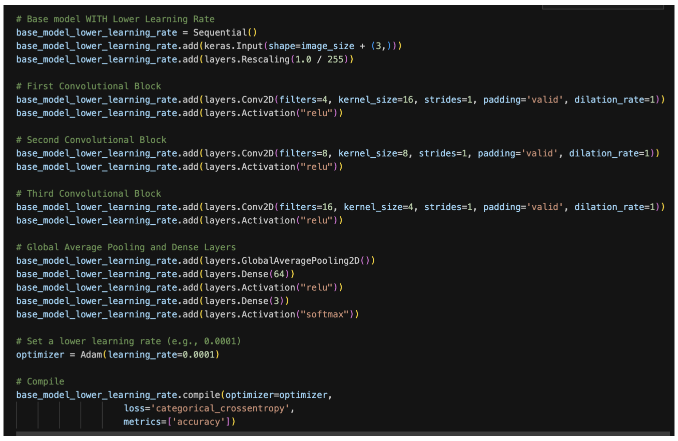
Fig. 7 Experiment 5 - Base Model with more Dilation Rate

5. **Modified Learning Rate:**
- Higher Learning Rate: 0.01 (default Adam is 0.001)
- Lower Learning Rate: 0.0001



```python
from tensorflow.keras.optimizers import Adam

# Base model WITH Higher Learning Rate
base_model_higher_learning_rate = Sequential()
base_model_higher_learning_rate.add(keras.Input(shape=image_size + (3,)))
base_model_higher_learning_rate.add(layers.Rescaling(1.0 / 255))

# First Convolutional Block
base_model_higher_learning_rate.add(layers.Conv2D(filters=4, kernel_size=16, strides=1, padding='valid', dilation_rate=1))
base_model_higher_learning_rate.add(layers.Activation("relu"))

# Second Convolutional Block
base_model_higher_learning_rate.add(layers.Conv2D(filters=8, kernel_size=8, strides=1, padding='valid', dilation_rate=1))
base_model_higher_learning_rate.add(layers.Activation("relu"))

# Third Convolutional Block
base_model_higher_learning_rate.add(layers.Conv2D(filters=16, kernel_size=4, strides=1, padding='valid', dilation_rate=1))
base_model_higher_learning_rate.add(layers.Activation("relu"))

# Global Average Pooling and Dense Layers
base_model_higher_learning_rate.add(layers.GlobalAveragePooling2D())
base_model_higher_learning_rate.add(layers.Dense(64))
base_model_higher_learning_rate.add(layers.Activation("relu"))
base_model_higher_learning_rate.add(layers.Dense(3))
base_model_higher_learning_rate.add(layers.Activation("softmax"))

# Set a higher learning rate (e.g., 0.01)
optimizer = Adam(learning_rate=0.01)
```

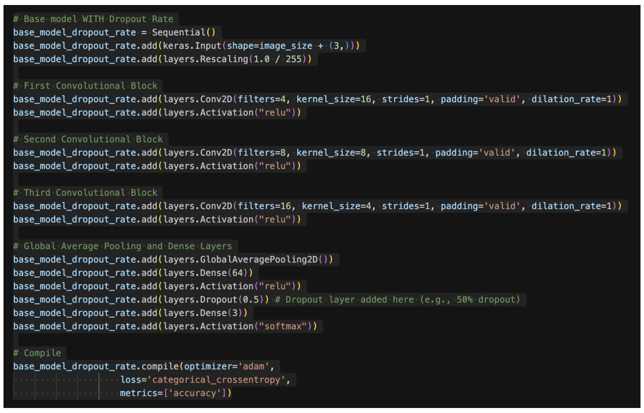
Fig. 8 Experiment 6 - Base Model with Higher learning rate



```python
# Base model WITH Lower Learning Rate
base_model_lower_learning_rate = Sequential()
base_model_lower_learning_rate.add(keras.Input(shape=image_size + (3,)))
base_model_lower_learning_rate.add(layers.Rescaling(1.0 / 255))

# First Convolutional Block
base_model_lower_learning_rate.add(layers.Conv2D(filters=4, kernel_size=16, strides=1, padding='valid', dilation_rate=1))
base_model_lower_learning_rate.add(layers.Activation("relu"))

# Second Convolutional Block
base_model_lower_learning_rate.add(layers.Conv2D(filters=8, kernel_size=8, strides=1, padding='valid', dilation_rate=1))
base_model_lower_learning_rate.add(layers.Activation("relu"))

# Third Convolutional Block
base_model_lower_learning_rate.add(layers.Conv2D(filters=16, kernel_size=4, strides=1, padding='valid', dilation_rate=1))
base_model_lower_learning_rate.add(layers.Activation("relu"))

# Global Average Pooling and Dense Layers
base_model_lower_learning_rate.add(layers.GlobalAveragePooling2D())
base_model_lower_learning_rate.add(layers.Dense(64))
base_model_lower_learning_rate.add(layers.Activation("relu"))
base_model_lower_learning_rate.add(layers.Dense(3))
base_model_lower_learning_rate.add(layers.Activation("softmax"))

# Set a lower learning rate (e.g., 0.0001)
optimizer = Adam(learning_rate=0.0001)

# Compile
base_model_lower_learning_rate.compile(optimizer=optimizer,
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

Fig.9 Experiment 7 - Base Model with Lower learning rate

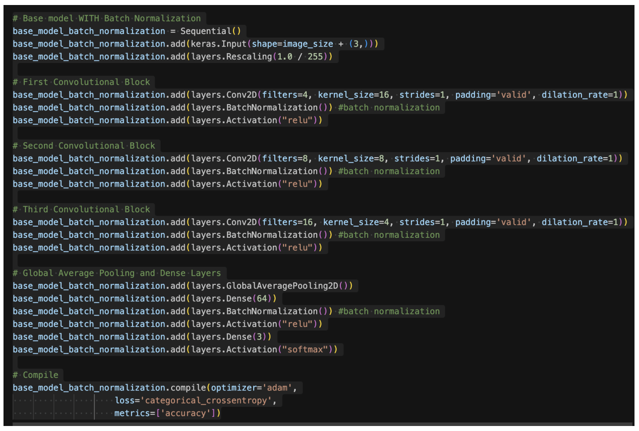
6. **Added Dropout:** 50% dropout rate after the dense layer



```python
# Base model WITH Dropout Rate
base_model_dropout_rate = Sequential()
base_model_dropout_rate.add(keras.Input(shape=image_size + (3,)))
base_model_dropout_rate.add(layers.Rescaling(1.0 / 255))

# First Convolutional Block
base_model_dropout_rate.add(layers.Conv2D(filters=4, kernel_size=16, strides=1, padding='valid', dilation_rate=1))
base_model_dropout_rate.add(layers.Activation("relu"))

# Second Convolutional Block
base_model_dropout_rate.add(layers.Conv2D(filters=8, kernel_size=8, strides=1, padding='valid', dilation_rate=1))
base_model_dropout_rate.add(layers.Activation("relu"))

# Third Convolutional Block
base_model_dropout_rate.add(layers.Conv2D(filters=16, kernel_size=4, strides=1, padding='valid', dilation_rate=1))
base_model_dropout_rate.add(layers.Activation("relu"))

# Global Average Pooling and Dense Layers
base_model_dropout_rate.add(layers.GlobalAveragePooling2D())
base_model_dropout_rate.add(layers.Dense(64))
base_model_dropout_rate.add(layers.Activation("relu"))
base_model_dropout_rate.add(layers.Dropout(0.5)) # Dropout layer added here (e.g., 50% dropout)
base_model_dropout_rate.add(layers.Dense(3))
base_model_dropout_rate.add(layers.Activation("softmax"))

# Compile
base_model_dropout_rate.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

Fig. 10 Experiment 6 - Base Model with Dropout Rate

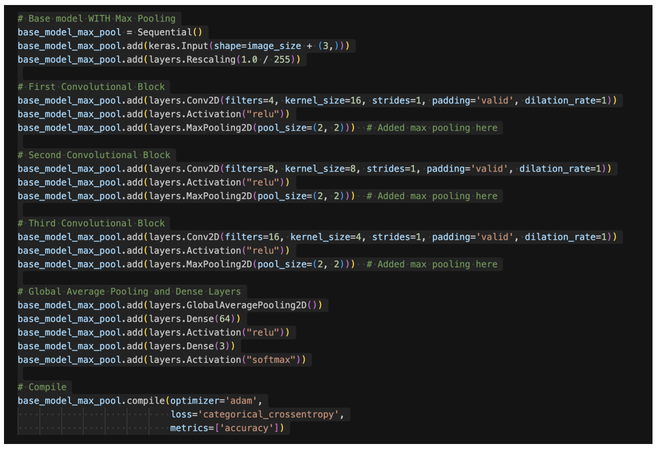
7. **Added Batch Normalization:** Applied after each convolutional and dense layer (before activation)



```python
# Base model WITH Batch Normalization
base_model_batch_normalization = Sequential()
base_model_batch_normalization.add(keras.Input(shape=image_size + (3,)))
base_model_batch_normalization.add(layers.Rescaling(1.0 / 255))

# First Convolutional Block
base_model_batch_normalization.add(layers.Conv2D(filters=4, kernel_size=16, strides=1, padding='valid', dilation_rate=1))
base_model_batch_normalization.add(layers.BatchNormalization()) #batch normalization
base_model_batch_normalization.add(layers.Activation("relu"))

# Second Convolutional Block
base_model_batch_normalization.add(layers.Conv2D(filters=8, kernel_size=8, strides=1, padding='valid', dilation_rate=1))
base_model_batch_normalization.add(layers.BatchNormalization()) #batch normalization
base_model_batch_normalization.add(layers.Activation("relu"))

# Third Convolutional Block
base_model_batch_normalization.add(layers.Conv2D(filters=16, kernel_size=4, strides=1, padding='valid', dilation_rate=1))
base_model_batch_normalization.add(layers.BatchNormalization()) #batch normalization
base_model_batch_normalization.add(layers.Activation("relu"))

# Global Average Pooling and Dense Layers
base_model_batch_normalization.add(layers.GlobalAveragePooling2D())
base_model_batch_normalization.add(layers.Dense(64))
base_model_batch_normalization.add(layers.BatchNormalization()) #batch normalization
base_model_batch_normalization.add(layers.Activation("relu"))
base_model_batch_normalization.add(layers.Dense(3))
base_model_batch_normalization.add(layers.Activation("softmax"))

# Compile
base_model_batch_normalization.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

Fig. 11 Experiment 7 - Base Model with Batch Normalization

8. **Added Max Pooling:** 2x2 max pooling after each convolutional block



```python
# Base model WITH Max Pooling
base_model_max_pool = Sequential()
base_model_max_pool.add(keras.Input(shape=image_size + (3,)))
base_model_max_pool.add(layers.Rescaling(1.0 / 255))

# First Convolutional Block
base_model_max_pool.add(layers.Conv2D(filters=4, kernel_size=16, strides=1, padding='valid', dilation_rate=1))
base_model_max_pool.add(layers.Activation("relu"))
base_model_max_pool.add(layers.MaxPooling2D(pool_size=(2, 2)))  # Added max pooling here

# Second Convolutional Block
base_model_max_pool.add(layers.Conv2D(filters=8, kernel_size=8, strides=1, padding='valid', dilation_rate=1))
base_model_max_pool.add(layers.Activation("relu"))
base_model_max_pool.add(layers.MaxPooling2D(pool_size=(2, 2)))  # Added max pooling here

# Third Convolutional Block
base_model_max_pool.add(layers.Conv2D(filters=16, kernel_size=4, strides=1, padding='valid', dilation_rate=1))
base_model_max_pool.add(layers.Activation("relu"))
base_model_max_pool.add(layers.MaxPooling2D(pool_size=(2, 2)))  # Added max pooling here

# Global Average Pooling and Dense Layers
base_model_max_pool.add(layers.GlobalAveragePooling2D())
base_model_max_pool.add(layers.Dense(64))
base_model_max_pool.add(layers.Activation("relu"))
base_model_max_pool.add(layers.Dense(3))
base_model_max_pool.add(layers.Activation("softmax"))

# Compile
base_model_max_pool.compile(optimizer='adam',
                            loss='categorical_crossentropy',
                            metrics=['accuracy'])
```

Fig. 12 Experiment 8 - Base Model with Max Pooling

IV. Results and Analysis

This section presents the experimental outcomes of the hyperparameter variations, including quantitative metrics and comparative analysis across models. The results highlight significant differences in model performance based on specific hyperparameter modifications.

A. *Training Performance*

The training metrics reveal several important insights about each hyperparameter's effect on model performance:

IV.A.1 Training Loss Comparison:

- Batch Normalization demonstrated the most significant reduction in training loss (approximately 0.66 by epoch 20)
- Base Model and Max Pooling both showed good convergence with steady decline in loss
- Larger Kernel Size and Dropout Rate exhibited the poorest loss reduction
- Lower Learning Rate showed slower loss reduction compared to other models

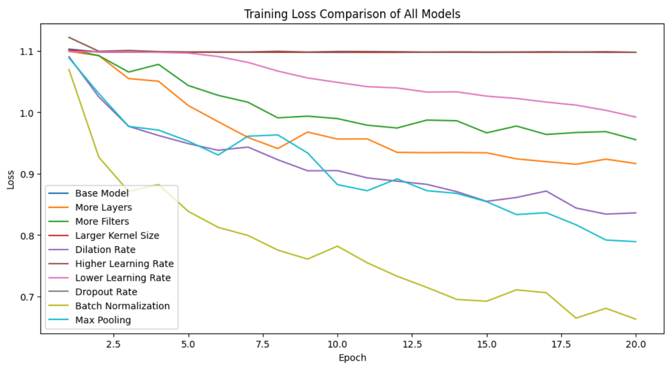


Fig. 13 Loss Comparison

IV.A.2 Training Accuracy Comparison:

- Batch Normalization achieved the highest training accuracy (approximately 71%)
- Max Pooling and Base Model both achieved respectable accuracy levels around 63-64%
- Higher Learning Rate and Dropout Rate showed the poorest training accuracy, remaining below 40%

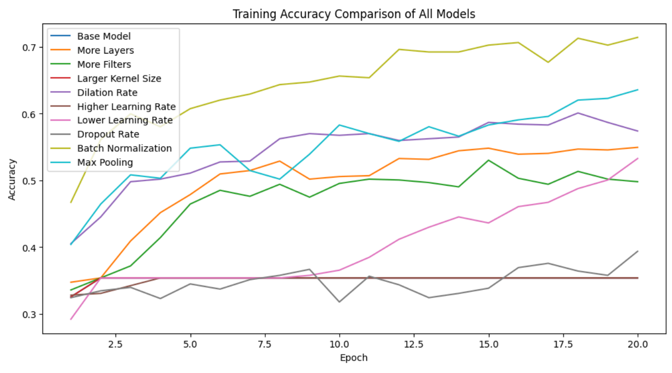


Fig. 14 Training Accuracy Comparison of All Models

B. *Validation Performance*

IV.B.1 Validation Loss Comparison:

- Most models exhibited fluctuating validation loss, indicating potential instability in generalization
- Dilation Rate and Batch Normalization models showed periods of lower validation loss
- Dropout Rate displayed significant spikes in validation loss
- Larger Kernel Size maintained relatively consistent, but high validation loss

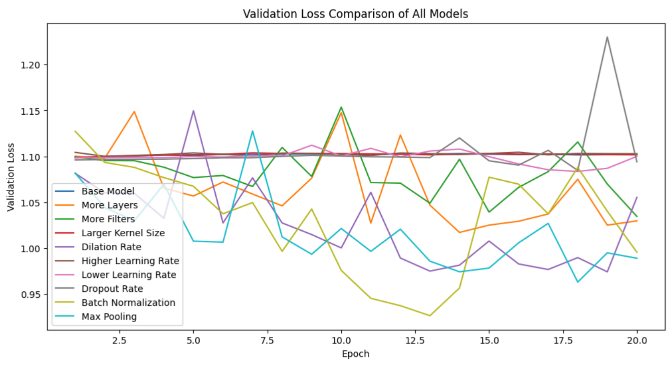


Fig. 15 Validation Loss Comparison

IV.B.2 Validation Accuracy Comparison:

- Batch Normalization reached the highest peak validation accuracy of around 58%
- Max Pooling and Dilation Rate also performed well, with validation accuracies consistently above 50% in later epochs
- Higher Learning Rate showed the poorest generalization, with validation accuracy stagnating at approximately 29%
- Lower Learning Rate showed a steady but slow improvement in validation accuracy

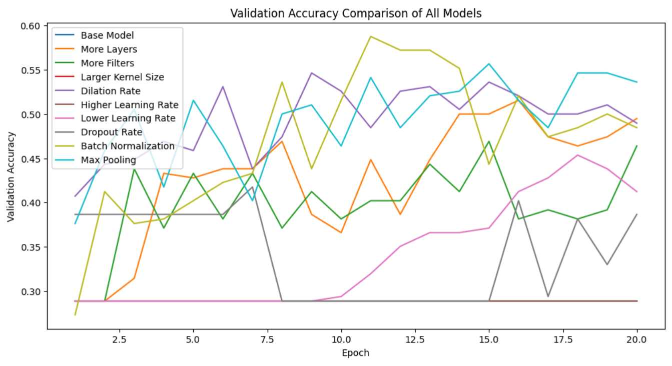


Fig. 16 Validation Accuracy Comparison of All Models

C. *Classification Results*

The models showed varying biases in their classification predictions:

TABLE I
Classification Results of Each Model Variation

Model Variation	**Curly Hair (%)**	**Straight Hair (%)**	**Wavy Hair (%)**
Base Model	35.18	31.72	33.10
More Layers	75.90	20.32	3.79

More Filters	65.23	22.46	12.31
Larger Kernel Size	35.18	31.53	33.29
Dilation Rate	90.00	8.28	1.72
Higher Learning Rate	35.44	31.74	32.82
Lower Learning Rate	72.07	15.26	12.66
Dropout Rate	28.90	34.28	36.83
Batch Normalization	67.91	29.30	2.78
Max Pooling	87.24	5.39	7.37

D. Key Findings

IV.D.1 Most Effective Hyperparameters:

- Batch Normalization: Provided the most significant improvement in both training and validation metrics, helping stabilize training and improve generalization
- Max Pooling: Offered strong performance gains while reducing computational complexity
- Dilation Rate: Showed promising results for feature detection at multiple scales

IV.D.2 Least Effective Hyperparameters:

- Higher Learning Rate: Led to poor convergence and generalization
- Dropout Rate: The chosen rate (0.5) may have been too aggressive for this dataset size
- Larger Kernel Size: Did not provide significant benefits and may have reduced the model's ability to capture fine details

IV.D.3 Class Bias:

- Many models showed a strong bias toward classifying images as "Curly" hair
- This bias suggests potential imbalances in the dataset or distinctive features in curly hair that are more easily recognized by the models

E. Comparative Analysis

The table presents a summary of the final validation accuracy for each model configuration.

TABLE II
Metrics Comparison of Both Models

Model Variation	**Final Validation Accuracy**
Base Model	~49%
More Layers	~48%
More Filter	~46%
Larger Kernel Size	~29%
Dilation Rate	~50%
Higher Learning Rate	~29%
Lower Learning Rate	~42%
Dropout Rate	~38%
Batch Normalization	~58%
Max Pooling	~54%

The results indicate that architectural modifications like batch normalization and max pooling provide the most substantial improvements over the baseline model. Batch normalization applied throughout the network (after each layer but before activation) effectively stabilized training across varying filter complexities ($4 \rightarrow 8 \rightarrow 16$), while the strategic 2×2 max pooling created an efficient feature hierarchy that preserved critical hair texture relationships (Ioffe & Szegedy, 2015).

These findings align with established principles in deep learning literature regarding the stabilizing effects of batch normalization in reducing internal covariate shift and the feature selection benefits of pooling operations in providing translation invariance for image classification tasks (Ioffe & Szegedy, 2015; Yu & Koltun, 2016). The combination proved particularly effective for distinguishing between subtle hair texture differences in the 64×64 pixel images used in this study.

V. Conclusion

This study has demonstrated the significant impact of hyperparameter selection on CNN performance for hair classification tasks. Batch normalization emerged as the most effective modification, substantially improving both convergence and generalization. Max pooling and dilation rate adjustments also proved beneficial, while larger kernel sizes and aggressive dropout rates were detrimental to performance.

The class distribution analysis revealed a tendency for many models to favor the "Curly" class, suggesting potential dataset imbalances that should be addressed in future work.

Based on these findings, we recommend the following directions for future research:

1. Combining successful hyperparameters, particularly batch normalization with moderate dilation rates and max pooling
2. Implementing class weighting to address prediction biases
3. Exploring data augmentation to increase dataset diversity
4. Investigating transfer learning approaches using pre-trained models
5. Testing alternative regularization techniques with reduced dropout rates
6. Implementing early stopping mechanisms to prevent overfitting

VI. References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2016, pp. 770-778.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in Advances in Neural Information Processing Systems, 2012, pp. 1097-1105.

[3] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in European Conf. Comput. Vis., 2014, pp. 818-833.

[4] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," in Int. Conf. Learn. Represent., 2016.

[5] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.

[6] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," J. Mach. Learn. Res., vol. 15, no. 1, pp. 1929-1958, 2014.

[7] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in Proc. Int. Conf. Mach. Learn., 2015, pp. 448-456.

[8] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in Int. Conf. Artif. Neural Netw., 2010, pp. 92-101.

[9] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. Cambridge, MA, USA: MIT Press, 2016.

[10] C. M. Bishop, Pattern Recognition and Machine Learning. New York, NY, USA: Springer, 2006.

[11] F. Chollet, Deep Learning with Python, 2nd ed. Shelter Island, NY, USA: Manning Publications, 2021.

[12] T. Hastie, R. Tibshirani, and J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd ed. New York, NY, USA: Springer, 2009.