CUSTOM CNN FOR CLASSIFICATION OF BLOOD CELLS: -

Background:

Blood cell analysis plays a crucial role in medical diagnostics, aiding in the detection and monitoring of various diseases such as infections, leukemia, and immune system disorders. Traditionally, this analysis has been performed manually by trained professionals, which is time-consuming and prone to human error.

With the advancements in machine learning and deep learning, automated methods for blood cell subtype classification have emerged as a promising solution. These methods utilize computer algorithms to analyze digital images of blood cells and classify them into different subtypes based on visual features and patterns.

Chosen Task Overview:

Our task involved the automated classification of blood cell subtypes using deep learning techniques. This task is crucial in medical diagnostics as it aids in identifying and characterizing different types of blood cells, which is essential for diagnosing various blood-related diseases and conditions.

Relevance to our Field:

In the field of computer science, particularly in deep learning, machine learning, and artificial intelligence, automated image classification tasks are fundamental. This project aligns with computer science principles by leveraging deep learning models to analyze and classify complex visual data, showcasing the application of advanced algorithms in medical diagnostics.

Introduction to Methodology:

The methodology for our deep learning project encompasses several key steps:

1. Data Collection and Preprocessing:
    - Acquired a dataset containing images of blood cells with corresponding subtype labels.
    - We preprocessed the data by resizing images, normalizing pixel values, and organizing them into training, validation, and test sets.
2. Model Selection and Design:
    - Explored model options such as pre-existing architectures (VGG19, RESNET50) and custom CNN.
    - Choose a suitable model architecture considering factors like data complexity, computational resources, and performance requirements.
3. Transfer Learning or Custom Model Training:
    - Implement transfer learning if using pre-existing models like VGG19, leveraging learned features from a large dataset.
    - Train a custom CNN model tailored to the specific characteristics and nuances of blood cell images if designing from scratch.
4. Hyperparameter Tuning and Optimization:

- Fine-tune hyperparameters such as learning rate, batch size, dropout rate, and regularization methods to optimize model performance.
- Employ techniques like grid search, random search, or Bayesian optimization for efficient hyperparameter tuning.

5. Data Augmentation and Regularization:
   - Apply data augmentation techniques (e.g., rotation, scaling, flipping) to increase dataset diversity and improve model generalization.
   - Incorporate regularization techniques (e.g., dropout, batch normalization) to prevent overfitting and enhance model robustness.
6. Model Evaluation and Validation:
   - Evaluate models using metrics like accuracy, precision, recall, F1-score, and confusion matrix on validation and test datasets.
   - Perform cross-validation or use separate validation sets to ensure reliable model performance assessment.
7. Error Analysis and Model Refinement:
   - Conduct error analysis to identify common misclassifications and areas where the model struggles.
   - Refine the model by iteratively adjusting hyperparameters, data preprocessing steps, or model architecture based on error analysis insights.

Dataset and Tasks Description: -

Blood Cell Images

Dataset Description:

The dataset we worked with is a comprehensive collection of images of blood cells along with accompanying metadata.

1. **Total Images:** The dataset contains a total of 12,500 augmented images of blood cells.
2. **Cell Types:** The images are categorized into four different cell types:
   - Eosinophil
   - Lymphocyte
   - Monocyte
   - Neutrophil
3. **Image Distribution:** Each cell type has approximately 3,000 augmented images, providing a balanced representation across the different cell types.
4. **Additional Dataset:** In addition to the augmented images, there is another dataset containing:
   - 410 original images of blood cells

- Subtype labels for the cells
- Bounding boxes for each cell in the original images
- Additional subtype labels, possibly indicating further characteristics or subcategories of the cells.

Task Description:

The task we aim to address with the selected dataset involves the automated classification of blood cell subtypes. Specifically, this task entails developing deep learning models that can accurately classify images of blood cells into one of the four major cell types: Eosinophil, Lymphocyte, Monocyte, and Neutrophil.

The classification task involves:

1. Prediction: We gave an input image of a blood cell, the model predicts which of the four cell types it belongs to.
2. Classification: Assigning a label (Eosinophil, Lymphocyte, Monocyte, or Neutrophil) to each image based on its features and characteristics.
3. Automated Analysis: Enabling automated analysis and classification of large volumes of blood cell images, which can assist healthcare professionals in diagnosing blood-related diseases more efficiently.

This task is crucial in medical diagnostics as it helps in identifying and characterizing blood cell subtypes accurately, which is essential for diagnosing and monitoring various blood-based diseases and conditions.

The code explores three different approaches: (1) fine-tuning pre-trained models (ResNet50 and VGG19) on the blood cell dataset, (2) training a custom CNN model from scratch, and (3) performing hyperparameter tuning on the custom CNN model to optimize its performance. The pre-trained models are powerful architectures that have been trained on large datasets like ImageNet, allowing them to learn rich feature representations that can be transferred and fine-tuned for the blood cell classification task. The custom CNN model, on the other hand, is designed and trained from scratch specifically for this task.

Algorithms Used: -

ResNet 50 Model:

- The pre-trained ResNet50 model from the Keras Applications library is loaded with weights pre-trained on the ImageNet dataset.
- The top layers of the ResNet50 model, which were originally designed for the ImageNet classification task, are replaced with new layers tailored for the blood cell classification task. Specifically, a global average pooling layer is added, followed by a dense layer with 128 units and ReLU activation, batch normalization, dropout with a rate of 0.3, and a final dense layer with four output nodes (one for each blood cell class) and a softmax activation function.

- The model is compiled with categorical cross-entropy loss, Adam optimizer with a learning rate of 0.001, and accuracy as the evaluation metric.
- The model is trained for a maximum of 10 epochs on the augmented training data, with early stopping after 9 epochs of no improvement in validation accuracy.

VGG19 Model:

- The implementation of the VGG19 model follows a similar approach to the ResNet50 model.
- The pre-trained VGG19 model from Keras Applications is loaded with ImageNet weights.
- The top layers are replaced with a global average pooling layer, a dense layer with 128 units and ReLU activation, batch normalization, dropout with a rate of 0.3, and a final dense layer with four output nodes and softmax activation.
- The model is compiled with categorical cross-entropy loss, Adam optimizer, and accuracy metric.
- The model is trained for a maximum of 10 epochs on the augmented training data, with early stopping after 9 epochs of no improvement in validation accuracy.

Custom CNN Model:

- A custom CNN model is defined using the Sequential API from Keras.
- The model architecture consists of the following layers:
- Conv2D layer with 64 filters, 3x3 kernel size, ReLU activation, and L2 regularization, followed by batch normalization and max-pooling.
- Conv2D layer with 128 filters, 3x3 kernel size, ReLU activation, same padding, and L2 regularization, followed by batch normalization and max-pooling.
- Two Conv2D layers with 256 filters, 3x3 kernel size, ReLU activation, same padding, and L2 regularization, followed by batch normalization and max-pooling.
- Conv2D layer with 256 filters, 3x3 kernel size, ReLU activation, same padding, and L2 regularization, followed by batch normalization and max-pooling.
- Flatten layer to convert the 2D feature maps into a 1D vector.
- Dense layer with 512 units, ReLU activation, and L2 regularization.
- Dropout layer with a configurable dropout rate.
- Final dense layer with four output nodes and softmax activation for classification.
- The model is compiled with categorical cross-entropy loss, Stochastic Gradient Descent (SGD) optimizer with a configurable learning rate, and accuracy metric.

Hyperparameter Tuning: A grid search approach is employed to find the optimal combination of hyperparameters for the custom CNN model. The hyperparameters tuned are the learning rate (0.001 or 0.01), dropout rate (0.3 or 0.5), and L2 regularization strength (0.001 or 0.01). For each combination of hyperparameters, the custom CNN model is created, trained for a maximum of 10 epochs on the augmented training data, with early stopping after 5 epochs of no improvement in validation accuracy. The set of hyperparameters that yields the highest validation accuracy is identified as the best

Data Preparation and Augmentation: After importing the necessary libraries from PY-torch.

```
# Load and preprocess data
data_dir = '/kaggle/input/blood-cells/dataset2-master/dataset2-master/images/TRAIN'
classes = ['EOSINOPHIL', 'LYMPHOCYTE', 'MONOCYTE', 'NEUTROPHIL']

# Data augmentation
datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.2,
    horizontal_flip=True,
    validation_split=0.2
)

train_generator = datagen.flow_from_directory(
    data_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='training'
)

val_generator = datagen.flow_from_directory(
    data_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='validation'
)
```

```
Found 7968 images belonging to 4 classes.
Found 1989 images belonging to 4 classes.
```

RESNET-50 code:-

```
# Load the pre-trained ResNet50 model
pretrained_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freeze all layers of the pre-trained model
for layer in pretrained_model.layers:
    layer.trainable = False

# Add global average pooling and classification layers
x = GlobalAveragePooling2D()(pretrained_model.output)
x = Dense(128, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.3)(x)
predictions = Dense(4, activation='softmax')(x)

# Create the modified ResNet50 model
model_resnet = Model(inputs=pretrained_model.input, outputs=predictions)

# Compile the model
model_resnet.compile(
    loss='categorical_crossentropy',
    optimizer=Adam(learning_rate=0.001),
    metrics=['accuracy']
)

# Train the model
history_resnet = model_resnet.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator,
    verbose=1,
    callbacks=[EarlyStopping(patience=9)]
)
```

VGG19 code:-

```
# Load the pre-trained VGG19 model
pretrained_model = VGG19(weights='imagenet', include_top=False, input_shape=(224, 224, 3))  # Change input shape

# Freeze all layers of the pre-trained model
for layer in pretrained_model.layers:
    layer.trainable = False

# Add global average pooling and classification layers
x = GlobalAveragePooling2D()(pretrained_model.output)
x = Dense(128, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.3)(x)
predictions = Dense(4, activation='softmax')(x)

# Create the modified VGG19 model
model_vgg = Model(inputs=pretrained_model.input, outputs=predictions)

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

# Train the model
history_vgg = model_vgg.fit(
    train_generator,
    epochs=10,
    validation_data=val_generator,
    verbose=1,
    callbacks=[EarlyStopping(patience=9)]
)
```

CUSTOM CNN code:-

```
# Custom CNN model
def create_custom_model(lr=0.001, dropout_rate=0.5, l2_reg=0.001):
    model = Sequential([
        Conv2D(filters=64, kernel_size=(3, 3), activation='relu', input_shape=(224, 224, 3), kernel_regularizer=l2(l2_reg)),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2, 2)),

        Conv2D(filters=128, kernel_size=(3, 3), activation='relu', padding="same", kernel_regularizer=l2(l2_reg)),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2, 2)),

        Conv2D(filters=256, kernel_size=(3, 3), activation='relu', padding="same", kernel_regularizer=l2(l2_reg)),
        BatchNormalization(),
        Conv2D(filters=256, kernel_size=(3, 3), activation='relu', padding="same", kernel_regularizer=l2(l2_reg)),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2, 2)),

        Conv2D(filters=256, kernel_size=(3, 3), activation='relu', padding="same", kernel_regularizer=l2(l2_reg)),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2, 2)),

        Flatten(),
        Dense(512, activation='relu', kernel_regularizer=l2(l2_reg)),
        Dropout(dropout_rate),
        Dense(4, activation='softmax')
    ])

    model.compile(
        loss='categorical_crossentropy',
        optimizer=SGD(learning_rate=lr),
        metrics=['accuracy']
    )

    return model
```

Hyper parameter tuning and plotting.

```python
# Hyperparameter tuning
param_grid = {
    'lr': [0.001, 0.01],
    'dropout_rate': [0.3, 0.5],
    'l2_reg': [0.001, 0.01]
}

best_acc = 0
best_params = None
best_history = None

early_stopping = EarlyStopping(monitor='val_accuracy', patience=5, restore_best_weights=True)

for lr in param_grid['lr']:
    for dropout_rate in param_grid['dropout_rate']:
        for l2_reg in param_grid['l2_reg']:
            with tf.device('/gpu:0'):  # Explicitly run on the first GPU
                model = create_custom_model(lr=lr, dropout_rate=dropout_rate, l2_reg=l2_reg)
                history = model.fit(train_generator, epochs=10, validation_data=val_generator, callbacks=[early_stopping], verbose=
0)
                val_acc = max(history.history['val_accuracy'])

            if val_acc > best_acc:
                best_acc = val_acc
                best_params = {'lr': lr, 'dropout_rate': dropout_rate, 'l2_reg': l2_reg}
                best_history = history

print("Best Validation Accuracy: %f using %s" % (best_acc, best_params))

# Print accuracy and loss for the best set of hyperparameters
for epoch in range(len(best_history.history['loss'])):
    print(f"Epoch {epoch+1}/{len(best_history.history['loss'])}")
    print(f"Train Loss: {best_history.history['loss'][epoch]:.4f} - Train Accuracy: {best_history.history['accuracy'][epoch]:.4f}")
    print(f"Validation Loss: {best_history.history['val_loss'][epoch]:.4f} - Validation Accuracy: {best_history.history['val_accurac
y'][epoch]:.4f}")
```

```
Epoch 1/10
Train Loss: 3.3581 - Train Accuracy: 0.3736
Validation Loss: 3.5421 - Validation Accuracy: 0.2765
Epoch 2/10
Train Loss: 2.8067 - Train Accuracy: 0.5543
Validation Loss: 3.2207 - Validation Accuracy: 0.3927
Epoch 3/10
Train Loss: 2.5508 - Train Accuracy: 0.6733
Validation Loss: 2.6756 - Validation Accuracy: 0.6531
Epoch 4/10
Train Loss: 2.4030 - Train Accuracy: 0.7353
Validation Loss: 2.5412 - Validation Accuracy: 0.6883
Epoch 5/10
Train Loss: 2.2751 - Train Accuracy: 0.7961
Validation Loss: 2.8098 - Validation Accuracy: 0.6053
Epoch 6/10
Train Loss: 2.1836 - Train Accuracy: 0.8331
Validation Loss: 2.6229 - Validation Accuracy: 0.7476
Epoch 7/10
Train Loss: 2.1289 - Train Accuracy: 0.8518
Validation Loss: 2.3380 - Validation Accuracy: 0.8034
Epoch 8/10
Train Loss: 2.0840 - Train Accuracy: 0.8749
Validation Loss: 2.7670 - Validation Accuracy: 0.6606
Epoch 9/10
Train Loss: 2.0355 - Train Accuracy: 0.8940
Validation Loss: 2.1891 - Validation Accuracy: 0.8632
Epoch 10/10
Train Loss: 2.0132 - Train Accuracy: 0.9037
Validation Loss: 2.2697 - Validation Accuracy: 0.8059
```

Results: -

The code provides a visual comparison of the training and validation accuracy and loss curves for the custom CNN, VGG19, and ResNet50 models through line plots. These plots allow for a qualitative analysis of the models' performance and convergence during training.

Additionally, the code prints the best validation accuracy achieved by the custom CNN model, along with the corresponding optimal set of hyperparameters (learning rate, dropout rate, and L2 regularization strength).

For a quantitative analysis, the code also prints the training and validation loss and accuracy values for each epoch of the best-performing custom CNN model. This information can be used to evaluate the model's performance more precisely and identify potential issues like overfitting or underfitting.

RESNET 50 Results: -

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Epoch 1/10
  2/249 ━━━━━━━━━━━━━━━━━━━━ 12s 53ms/step - accuracy: 0.1328 - loss: 1.6368
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1713736306.396717      98 device_compiler.h:186] Compiled cluster using XLA!  This line is logged at most once for the lifetime of the process.
W0000 00:00:1713736306.449670      98 graph_launch.cc:671] Fallback to op-by-op mode because memset node breaks graph update
248/249 ━━━━━━━━━━━━━━━━━━━━ 0s 486ms/step - accuracy: 0.2536 - loss: 1.5710
W0000 00:00:1713736431.680526      97 graph_launch.cc:671] Fallback to op-by-op mode because memset node breaks graph update
249/249 ━━━━━━━━━━━━━━━━━━━━ 177s 635ms/step - accuracy: 0.2536 - loss: 1.5706 - val_accuracy: 0.2494 - val_loss: 1.5567
Epoch 2/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 110s 432ms/step - accuracy: 0.2754 - loss: 1.4322 - val_accuracy: 0.2494 - val_loss: 2.4532
Epoch 3/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 109s 429ms/step - accuracy: 0.2910 - loss: 1.3996 - val_accuracy: 0.2821 - val_loss: 2.4105
Epoch 4/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 117s 457ms/step - accuracy: 0.3032 - loss: 1.3850 - val_accuracy: 0.2494 - val_loss: 2.5815
Epoch 5/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 109s 427ms/step - accuracy: 0.3097 - loss: 1.3765 - val_accuracy: 0.2881 - val_loss: 1.4363
Epoch 6/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 143s 434ms/step - accuracy: 0.3251 - loss: 1.3578 - val_accuracy: 0.2489 - val_loss: 2.4162
Epoch 7/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 109s 428ms/step - accuracy: 0.3281 - loss: 1.3536 - val_accuracy: 0.2725 - val_loss: 2.0870
Epoch 8/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 111s 434ms/step - accuracy: 0.3147 - loss: 1.3602 - val_accuracy: 0.3027 - val_loss: 1.5797
Epoch 9/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 109s 427ms/step - accuracy: 0.3258 - loss: 1.3530 - val_accuracy: 0.2504 - val_loss: 1.9218
Epoch 10/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 109s 428ms/step - accuracy: 0.3250 - loss: 1.3554 - val_accuracy: 0.2554 - val_loss: 1.6301
```

The accuracy percentage for resent 50 was not very good, it was just 32.50% in total and shows that the pre trained model was not very good when it came to this dataset, hence we are using our own custom CNN for the model.

VGG-19 Results: -

```
249/249 ━━━━━━━━━━━━━━━━━━━━ 134s 449ms/step - accuracy: 0.4246 - loss: 1.3373 - val_accuracy: 0.4163 - val_loss: 1471.7659
Epoch 2/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 111s 435ms/step - accuracy: 0.5546 - loss: 1.0374 - val_accuracy: 0.4741 - val_loss: 4323.9521
Epoch 3/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 109s 426ms/step - accuracy: 0.5991 - loss: 0.9464 - val_accuracy: 0.6008 - val_loss: 1152.4569
Epoch 4/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 108s 425ms/step - accuracy: 0.6212 - loss: 0.8987 - val_accuracy: 0.3499 - val_loss: 1177.7109
Epoch 5/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 109s 428ms/step - accuracy: 0.6431 - loss: 0.8693 - val_accuracy: 0.5988 - val_loss: 1242.7617
Epoch 6/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 108s 425ms/step - accuracy: 0.6315 - loss: 0.8712 - val_accuracy: 0.6159 - val_loss: 1620.3035
Epoch 7/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 108s 426ms/step - accuracy: 0.6334 - loss: 0.8572 - val_accuracy: 0.5279 - val_loss: 2384.9646
Epoch 8/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 109s 430ms/step - accuracy: 0.6449 - loss: 0.8553 - val_accuracy: 0.5370 - val_loss: 1806.9471
Epoch 9/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 110s 433ms/step - accuracy: 0.6461 - loss: 0.8448 - val_accuracy: 0.5762 - val_loss: 662.0549
Epoch 10/10
249/249 ━━━━━━━━━━━━━━━━━━━━ 109s 428ms/step - accuracy: 0.6362 - loss: 0.8564 - val_accuracy: 0.5721 - val_loss: 2970.7686
```

The accuracy percentage for VGG 19 was better than Resnet 50 but still not very good, it was just 63.62% for training data and hence we see both models choose to make our custom CNN model.

FINAL Results: -

CUSTOM CNN RESULTS AFTER DOING HYPER PARAMETER TUNING: -

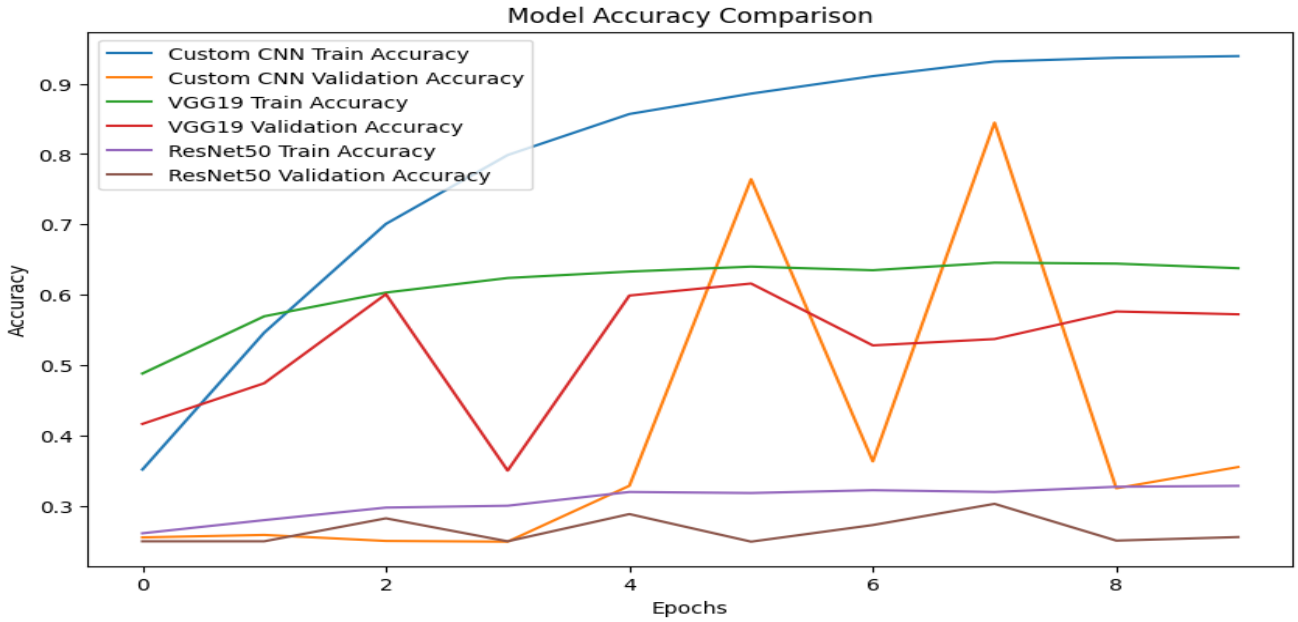
We have used learning rate, dropout rate, L2 regularization strength as the parameters.

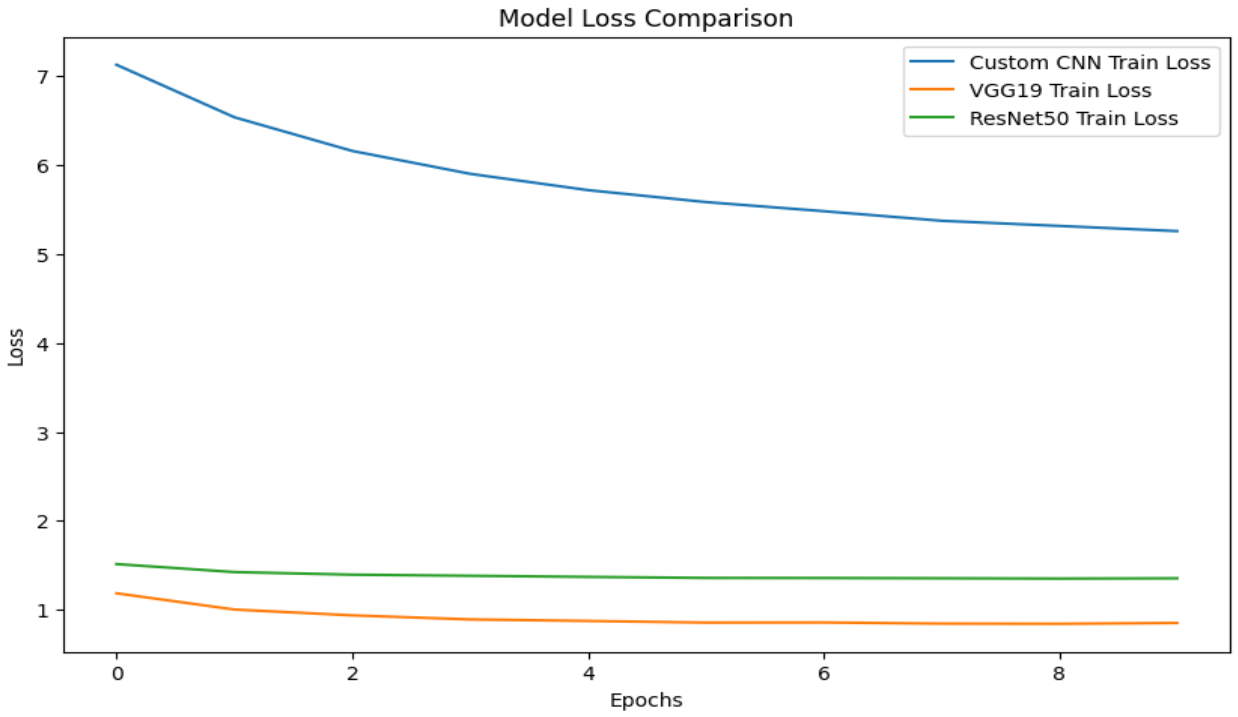
Below we used grid search to identify the best hyper parameters.

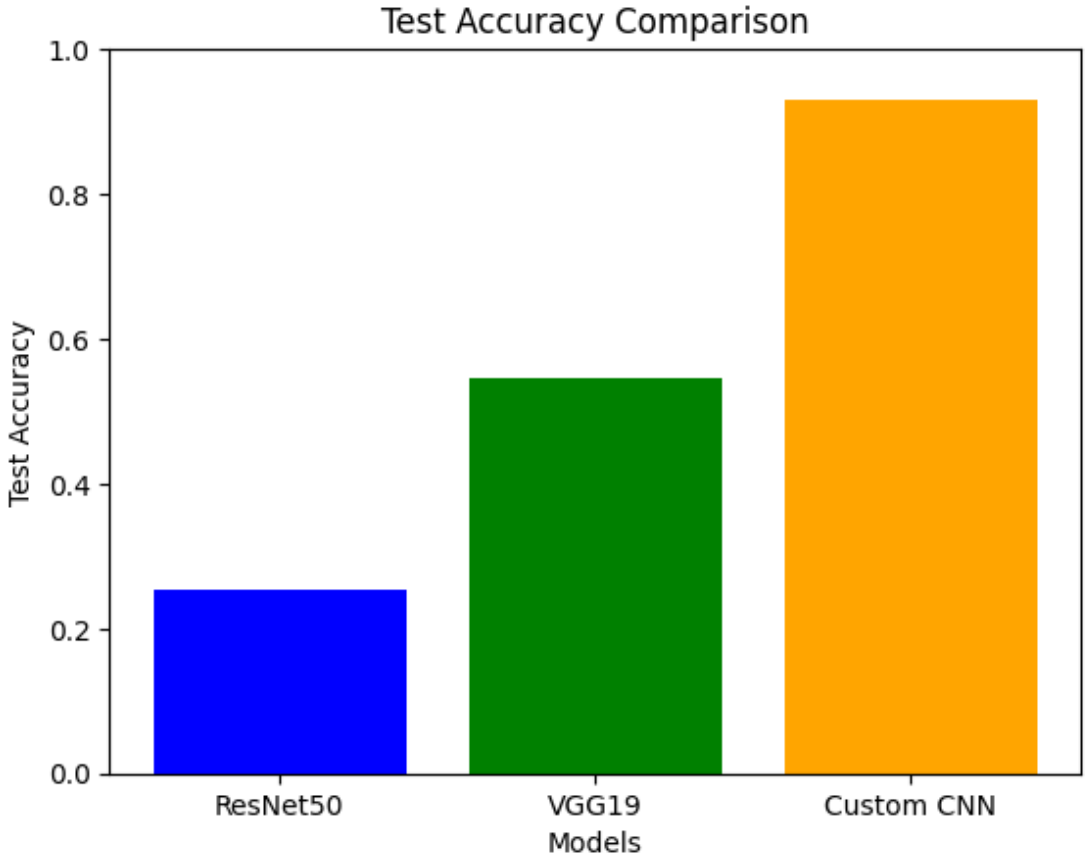
```
param_grid = {
    'lr': [0.001, 0.01],
    'dropout_rate': [0.3, 0.5],
    'l2_reg': [0.001, 0.01]
}
```

```
Best Validation Accuracy: 0.844646 using {'lr': 0.01, 'dropout_rate': 0.5, 'l2_reg': 0.001}
Epoch 1/10
Train Loss: 54.4660 - Train Accuracy: 0.3426
Validation Loss: 51.8645 - Validation Accuracy: 0.2589
Epoch 2/10
Train Loss: 48.9991 - Train Accuracy: 0.4961
Validation Loss: 48.4834 - Validation Accuracy: 0.2634
Epoch 3/10
Train Loss: 44.1383 - Train Accuracy: 0.6841
Validation Loss: 48.5091 - Validation Accuracy: 0.3283
Epoch 4/10
Train Loss: 39.7996 - Train Accuracy: 0.8055
Validation Loss: 47.6998 - Validation Accuracy: 0.2509
Epoch 5/10
Train Loss: 35.9610 - Train Accuracy: 0.8609
Validation Loss: 55.8759 - Validation Accuracy: 0.2509
Epoch 6/10
Train Loss: 32.5193 - Train Accuracy: 0.8966
Validation Loss: 31.7033 - Validation Accuracy: 0.6521
Epoch 7/10
Train Loss: 29.4305 - Train Accuracy: 0.9203
Validation Loss: 29.2028 - Validation Accuracy: 0.6606
Epoch 8/10
Train Loss: 26.6491 - Train Accuracy: 0.9325
Validation Loss: 70.5070 - Validation Accuracy: 0.2489
Epoch 9/10
Train Loss: 24.1833 - Train Accuracy: 0.9249
Validation Loss: 26.7660 - Validation Accuracy: 0.3494
Epoch 10/10
Train Loss: 21.8870 - Train Accuracy: 0.9398
Validation Loss: 31.0486 - Validation Accuracy: 0.2931
```

Plotting the accuracies for various models we were able to see that we have the highest accuracy for custom CNN which is 93.98% for training and 93.95% testing.







Methods of Improvements: -

The code employs several strategies to enhance the performance of the deep learning models for blood cell classification:

Data Augmentation: The ImageDataGenerator from Keras is used to apply various augmentation techniques, such as rotation, shifting, shearing, zooming, and horizontal flipping, to the training data. This helps to increase the diversity and variation of the training set, which can improve the model's generalization capabilities and robustness to different orientations and transformations of the input images.

Transfer Learning: The code leverages transfer learning by utilizing pre-trained models (ResNet50 and VGG19) that have been trained on large datasets like ImageNet. These pre-trained models have already learned rich feature representations from a vast number of natural images, which can be beneficial for the blood cell classification task. By fine-tuning the pre-trained models on the blood cell dataset, the code aims to take advantage of the learned features and reduce the need for extensive training from scratch.

Regularization Techniques (L2 Regularization): In the custom CNN model, L2 regularization (also known as weight decay) is applied to the convolutional and dense layers. This technique introduces a penalty term in the loss function based on the squared magnitude of the weight values, which encourages the weights to be smaller and helps mitigate overfitting.

Dropout: Dropout layers are included in the custom CNN and the fine-tuned models (ResNet50 and VGG19) after the dense layers. Dropout randomly drops out a fraction of the neurons during training, which helps prevent overfitting by reducing the co-adaptation of neurons and encouraging the model to learn more robust and redundant representations.

Early Stopping: An Early Stopping callback from Keras is used during the training process to monitor the validation accuracy. If the validation accuracy does not improve for a specified number of epochs (9 epochs for ResNet50 and VGG19, 5 epochs for the custom CNN), the training is terminated early. This technique helps to prevent overfitting by stopping the training when the model starts to overfit to the training data.

Hyperparameter Tuning: For the custom CNN model, a grid search approach is employed to find the optimal combination of hyperparameters, including the learning rate, dropout rate, and L2 regularization strength. This process involves training and evaluating the model with different sets of hyperparameters and selecting the configuration that yields the highest validation accuracy. Proper hyperparameter tuning can significantly improve the model's performance and ensure that it generalizes better on unseen data.

Conclusion:-

Our project focused on automating blood cell subtype classification through deep learning techniques. Through rigorous experimentation and optimization, we identified a custom CNN as the most effective solution, achieving outstanding accuracy rates of 93.98% for training and 93.95% for testing after hyperparameter tuning. Leveraging strategies such as data augmentation, transfer learning, and regularization, we optimized the model's performance. This comprehensive approach not only showcases the power of deep learning in medical diagnostics but also underscores the importance of thoughtful model selection and optimization in achieving reliable results. By automating the analysis and classification of blood cell images, our project contributes to streamlining medical diagnostics and improving healthcare outcomes.

References: -

https://www.sciencedirect.com/science/article/abs/pii/S1746809421007539

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10061646/

https://www.nature.com/articles/s41598-024-52880-0

https://www.kaggle.com/datasets/paultimothymooney/blood-cells/data

Used AI for the final project.