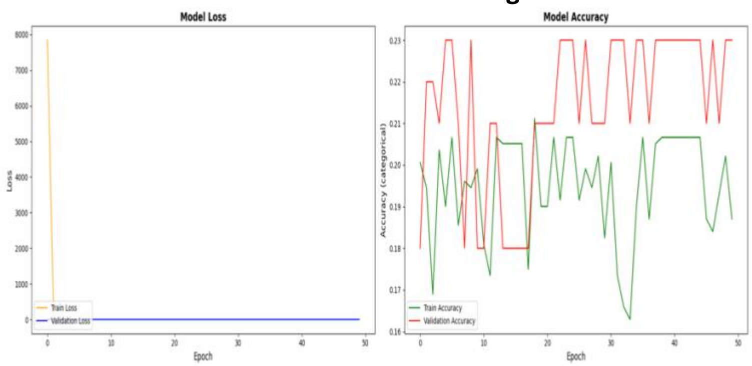
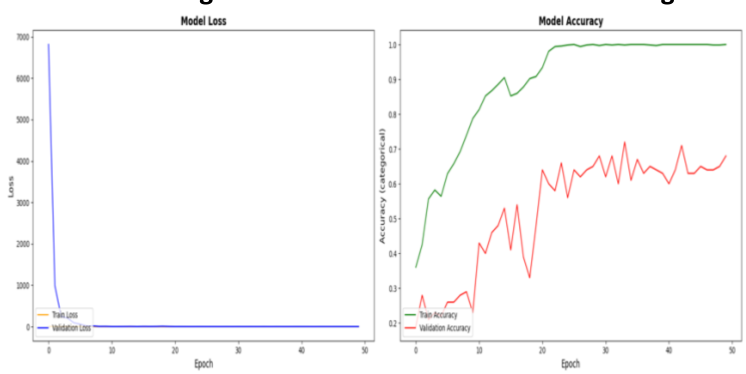
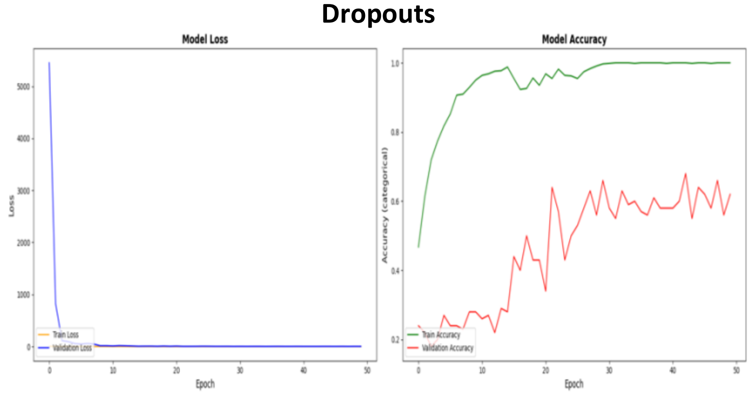
Gesture Recognition RNN write up

<table>
<tr><th>#</th><th>Model</th><th>Result</th><th>Observation + Decision + Explanation</th></tr>
<tr><td>1</td><td>Conv3D with MaxPooling3D

</td><td>Accuracy:
23%

Loss:
1.6068</td><td>The rapid decrease in training loss combined with minimal change in validation loss is a clear indicator of overfitting.

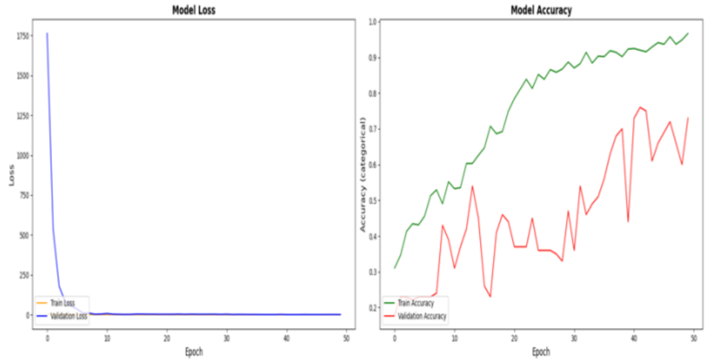
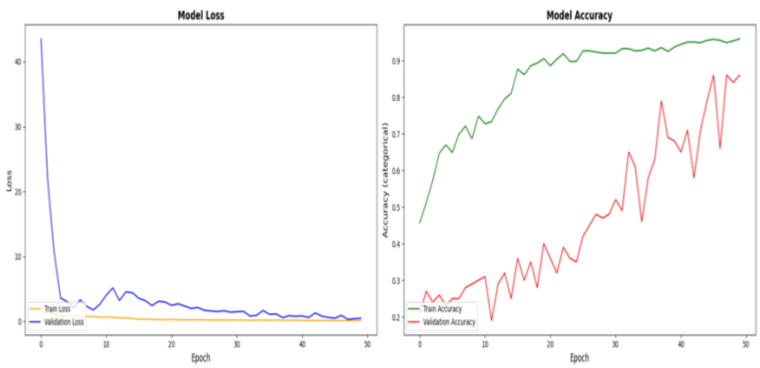
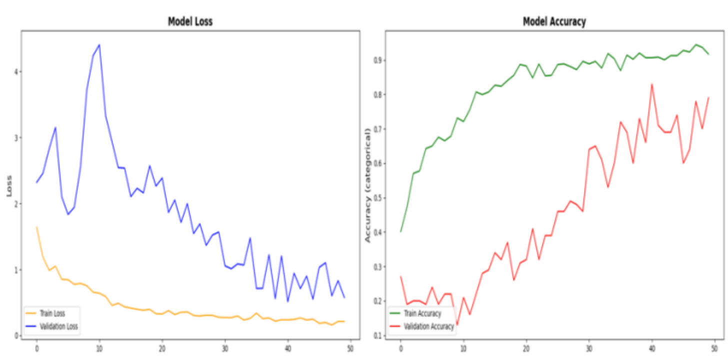
Decision for next step: Have decided to use reduce overfitting, by introducing Batch Normalization</td></tr>
<tr><td>2</td><td>Conv3D using BatchNormalization with MaxPooling3D

</td><td>Accuracy:
68%

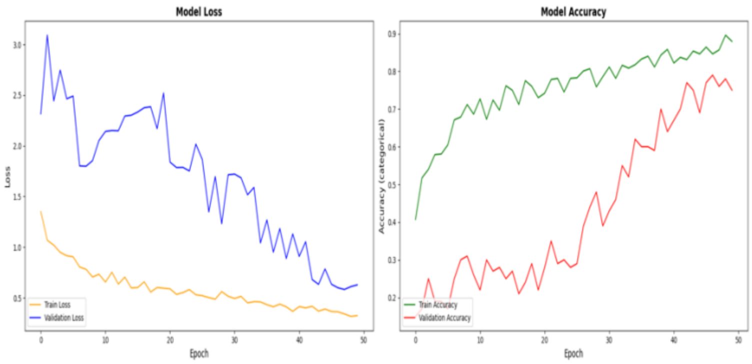
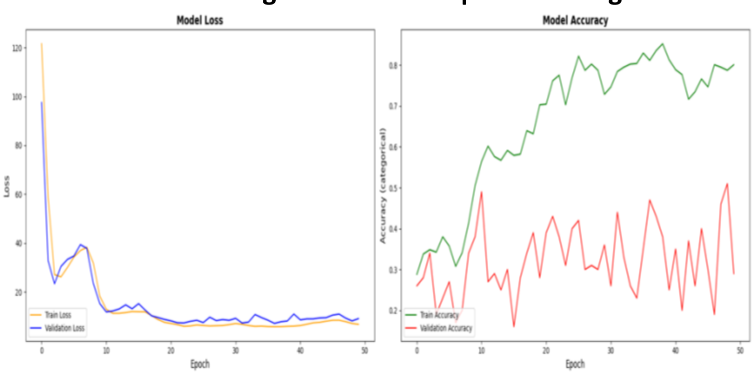
Loss:
1.4995</td><td>The large gap between training accuracy (nearly 100%) and validation accuracy (fluctuating at ~60%) indicates that the model is most likely overfitting to the training data. The fluctuations in the validation accuracy suggest that the model struggles to generalize well to new data.

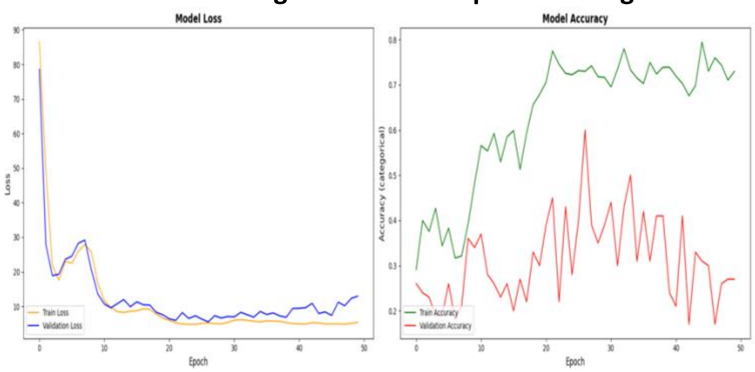
Decision for next step: Have decided to further improve overfitting issue, by introducing Dropouts</td></tr>
<tr><td>3</td><td>Conv3D with MaxPooling3D with Batch Normalization and Dropouts

</td><td>Accuracy:
62%

Loss:
2.2684</td><td>The model overfits the training data. This is evidenced by the stable and near-perfect train accuracy and low training loss, but significantly lower and noisy validation accuracy. The model learns well from the training data but cannot generalize well to validation data.

Decision for next step: Have decided to further improve overfitting issue, by using a different pooling approach like GlobalAveragePooling3D</td></tr>
</table>

4	**Conv3D with MaxPooling3D with Batch Normalization and Dropouts using GlobalAveragePooling3D** 	**Accuracy: 73%** **Loss: 0.8379**	While the loss curve looks great, the wide gap between training and validation accuracy, along with validation accuracy fluctuation, indicates overfitting. Regularization, early stopping, or learning rate tuning are possible solutions to boost the performance on unseen data and achieve better generalization. **Decision for next step:** **In the next step we have decided to use Conv2D in place of Conv3D to ensure further performance improvement**
5	**TimeDistributed Conv2D + Dense** 	**Accuracy: 86%** **Loss: 0.4855**	The model is exhibiting signs of overfitting as seen in the steep rise in training accuracy and less improvement in validation accuracy. Possible actions to improve generalization include decreasing the learning rate, adding regularization techniques such as dropout, and possibly increasing the training time (number of epochs). **Decision for next step:** **we have decided to use GRU to further speed up sequence processing with RNN**
6	**TimeDistributed Conv2D + GRU** 	**Accuracy: 79%** **Loss: 0.5741**	The model is learning and performing well on the training set However, the performance on the validation set is significantly worse, indicating overfitting and poor generalization. Applying the suggestions above, such as reducing model complexity, implementing early stopping, and tweaking the hyperparameters (e.g., learning rate), could potentially address these issues. **Decision for next step:** **we have decided to use ConvLSTM2D to further benchmark performance using RNN**

7	**TimeDistributed + ConvLSTM2D** 	**Accuracy: 75%** **Loss: 0.6279**	The model demonstrates the ability to learn and reduce error on the training data, but its generalization to the validation set is not optimal. While the loss metrics are decreasing, the accuracy gap between training and validation indicates overfitting. Adjusting complexity, introducing regularization, or expanding your dataset can help in achieving a better overall result. **Decision for next step: Next will be experimenting with Transfer Learning with less layers and all trainable params and LSTM network**
8	**Transfer Learning with trainable params using LSTM** 	**Accuracy:29%** **Loss: 9.1494**	Overfitting: The sharp decrease in training loss coupled with a fluctuating and flat validation accuracy suggests that the model is overfitting to the training data after around 10 epochs. Indicators: Large gap between the training and validation accuracy curves. Validation accuracy oscillates a lot, making it hard to achieve consistent generalization. 📉 Suggestions applied to Improve Generalization: Increase Regularization: The LSTM already has L2 regularization, but we can enhance this or introduce further dropout layers to increase the regularization, which might help control overfitting. Regularization term (0.01) used in the LSTM layer might need to be fine-tuned or increased. Reduce Model Complexity: The model might be too complex for the available amount of training data. One approach could be to reduce the capacity of the LSTM layer or reduce the number of parameters in the final fully connected layers.

			Decision for next step: **Next will be experimenting with Transfer Learning with less layers and all trainable params and GRU network and noticed like size of the model is getting higher**
9	**Transfer Learning with trainable params using GRU** 	**Accuracy: 27%** **Loss: 12.9480**	From the current behavior of the loss and accuracy graphs, it's clear that the model is overfitting and not generalizing well to the validation data. Introducing stronger regularization tactics and considering simpler architectures may help the model improve its performance on unseen data (validation). **Decision for next step:** **We decided like training all params has taken more time and we could not add more RNN layers to effectively learn the sequence features and noticed like size of the model is getting higher**
Final Model	**TimeDistributed Conv2D + Dense / TimeDistributed Conv2D + GRU**	**Accuracy: 86% / 79%** **Loss: .4855 / .5741**	Based on the analysis on the results from our experiments we decided like to be used in the webcam for TV to monitor gesture recognition use case we wanted a high performance and low memory footprint model. **And our TimeDistributed Conv2D + Dense/GRU and even ConvLSTM2D came very close to meeting our expectations, so hence we decided to rank and use Conv2D + RNN as the better choice based on our experiments**

Model rankings based on final validation accuracy:

Rank 1: Model 5 with validation accuracy = 0.8600
Rank 2: Model 6 with validation accuracy = 0.7900
Rank 3: Model 7 with validation accuracy = 0.7500
Rank 4: Model 4 with validation accuracy = 0.7300
Rank 5: Model 2 with validation accuracy = 0.6800
Rank 6: Model 3 with validation accuracy = 0.6200
Rank 7: Model 8 with validation accuracy = 0.2900
Rank 8: Model 9 with validation accuracy = 0.2700
Rank 9: Model 1 with validation accuracy = 0.2300

Model rankings based on final validation loss:

Rank 1: Model 5 with validation loss = 0.4855
Rank 2: Model 6 with validation loss = 0.5741
Rank 3: Model 7 with validation loss = 0.6279
Rank 4: Model 4 with validation loss = 0.8379
Rank 5: Model 2 with validation loss = 1.4995
Rank 6: Model 1 with validation loss = 1.6068
Rank 7: Model 3 with validation loss = 2.2684
Rank 8: Model 8 with validation loss = 9.1494
Rank 9: Model 9 with validation loss = 12.9480

Model Summaries

	Model	Total Parameters	Trainable Parameters	Non-Trainable Parameters	Number of Layers	Model Size (MB)
0	Conv3D_MaxPooling3D	8311813	8311813	0	13	31.71
1	Conv3D_BatchNormalization_MaxPooling3D	8317701	8314757	2944	18	31.73
2	Conv3D_BatchNormalization_Dropouts_MaxPooling3D	22732549	22730629	1920	17	86.72
3	Conv3D_BatchNormalization_Dropouts_GlobalAvera...	712453	710533	1920	17	2.72
4	TimeDistributed_Conv2D_Dense	129477	128517	960	13	0.49
5	TimeDistributed_Conv2D_GRU	99845	99269	576	15	0.38
6	TimeDistributed_ConvLSTM2D	13781	13589	192	11	0.05
7	Transfer_Learning_with_LSTM	32196357	32196101	256	10	122.82
8	Transfer_Learning_with_GRU	24152069	24151813	256	10	92.13

Note: in the above table the models are indexed from 0.

Final Result based on our experimented models

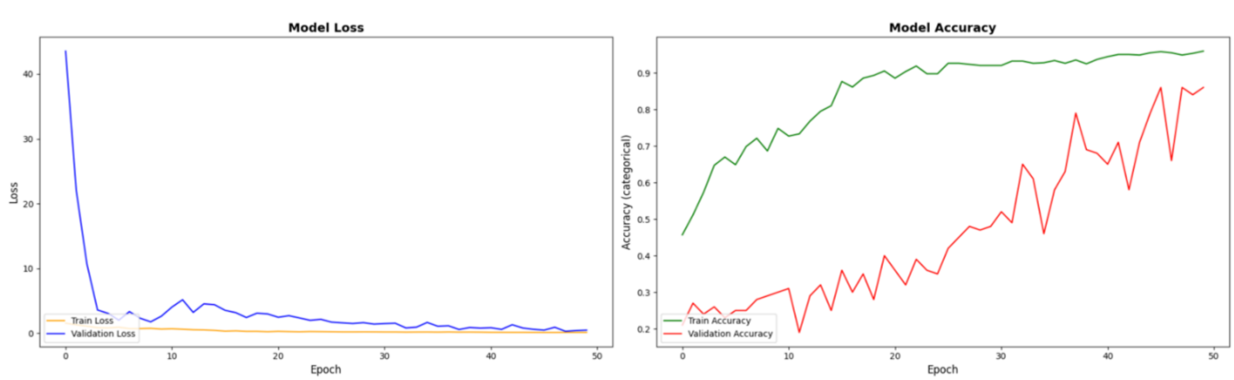
Based on the provided ranking tables and graphs for both validation loss and categorical accuracy, **Model 5 & 6** appears to be the best models.

Here's why Model 5 & 6 stands out:

1. **Best Validation Loss:**
    - Model 5 & 6 has the **lowest** validation loss at **0.4855 & 0.5741**, which is ranked 1st.
2. **Best Validation Accuracy:**
    - Model 5 * 6 also has the **highest** validation accuracy at **0.8600 & 0.7900**, topping the accuracy ranking as well.
3. **Consistent Performance in Accuracy Graph:**
    - Referring to the categorical accuracy graph:
        - The **green solid line** (Model 5 & 6 Training Accuracy) and the **green dashed line** (Model 5 Validation Accuracy) show a consistently **increasing trend**.
        - At the end of training, Model 5 & 6 has the highest validation accuracy compared to the other models, demonstrating **robust generalization**.
4. **Low Model Size and Parameters:**
    - Model 6, TimeDistributed_Conv2D_GRU and Model 5 TimeDistributed_Conv2D_Dense, has **relatively fewer parameters** (99,845 total, with 99,269 being trainable & 129477 totall, with 128517 being trainable).
    - This makes it **efficient for computation**, with notably smaller model size (0.38 MB & 0.4900) compared to much larger models like Model 8 (122.82 MB).

Conclusion:

Model 5 & 6 checks all the important aspects in terms of accuracy, loss, efficiency, and computational cost. Therefore, **Model 6** (TimeDistributed_Conv2D_GRU) and **Model 5** (TimeDistributed_Conv2D_Dense) is the best models based on the given data and the results from the experiments performed.



Future Experiments to improve

Based on the analysis provided for Model 6 (TimeDistributed_Conv2D_GRU) & Model 5 (TimeDistributed_Conv2D_Dense), while it performs well in terms of accuracy and model efficiency, we could explore a few modifications or entirely new models to further improve performance. Below are a few more model suggestions and areas for potential improvement in future study:

---

1. Incorporate Efficient Transfer Learning Model

- **Proposed New Model:** Transfer_Learning_with_EfficientNet + GRU/ConvLSTM combining time dependencies
- **Improvement Direction:**
    - Replace the Convolutional feature extraction in the time-distributed layers with a pre-trained model like **EfficientNet** or **MobileNet** for better feature extraction.
    - This transfer learning model will likely provide **more robust feature representations**, especially if your data is somewhat related to natural images. Once you have these features, you can apply GRU (or ConvLSTM) to capture the temporal dependencies (similar structure to Model 5/6).
- **Expected Benefits:**
    - You should experience faster convergence and improved accuracy (due to the powerful feature extraction capabilities of the transfer learning model).
    - This approach reduces the need to completely train the convolutional layers from scratch and may also reduce overfitting.
- **Compensation:**
    - The size of the transfer learning model might slightly increase, but you can choose compact transfer learning models like **MobileNet** to keep it efficient.

---

2. Increase Network Depth for Model 5

- **Proposed Alteration to Model 6:** Add **more GRU layers** or switch to a **Bi-directional GRU** with layer stacking.
    - **Improvement Direction:**
        - Increase the GRU capacity in Model 6 by stacking a few more GRU layers (or switch to **LSTM** depending on the data characteristics).
        - Experiment with **Bidirectional GRU/LSTM**, which helps the model capture **both forward and backward temporal dependencies** in the sequence.

- **Expected Benefits:**
  - This can enable **richer sequential pattern recognition** and potentially improve performance on datasets where long-range dependencies are crucial.
- **Compensation:**
  - Model complexity (in terms of parameters) will increase. Proper regularization (like dropout or L2 regularization) will be important to avoid overfitting.

5. Model Ensemble

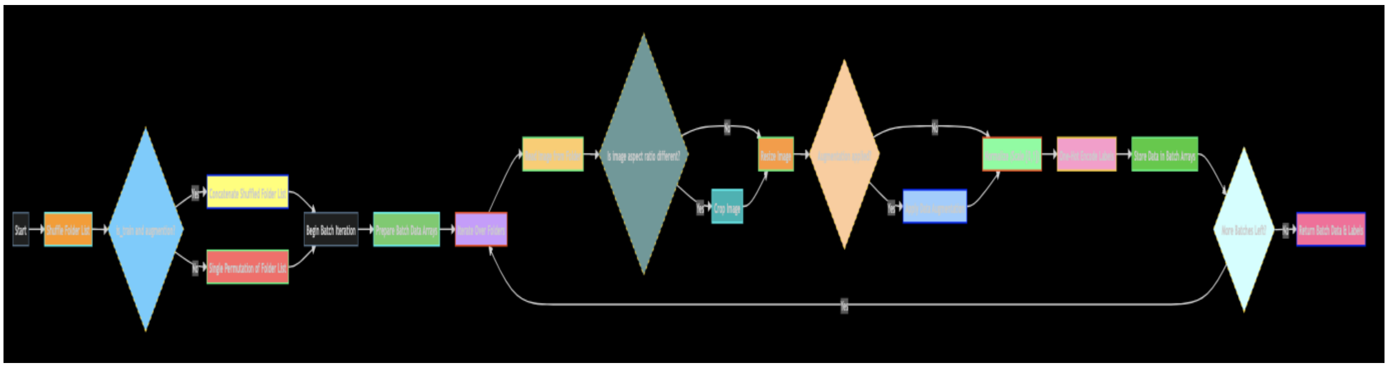
- **Proposed Improvement:** Combine multiple models via **Model Ensembling** (e.g., Ensemble of Model 6 + Model 7).
  - **Improvement Direction:**
    - To leverage the distinct strengths of different models, you may fuse predictions from **multiple models** (e.g., averaging or weighted ensemble of **Model 6** and **Model 7**, or other models with different architecture designs).
    - For instance, **CNN+LSTM** and **CNN+GRU** models may generalize differently, and ensemble learning can capture more diverse patterns in data.
  - **Expected Benefits:**
    - This can improve robustness and boost performance via a **wisdom of the crowd** approach.
    - The ensemble method can reduce variance (through model averaging/shrinking) and mitigate overfitting.
  - **Compensation:**
    - Increased computational cost due to multiple models, but ensemble approaches bring significant benefits when computational resources are not extremely constrained.

Conclusion:

- **Immediate Recommendations:**
  - **Stacking Bi-directional GRU layers** or adding **Attention** to Model 6 can be a **quick win** to experiment with without adding too much computational overhead.
- Additionally, **model ensemble techniques** can also leverage multiple architectures and boost robustness while taking a slight computational trade-off.

Generator Function Usage:

Python generator function named generator that is used for real-time **batch data generation** when training or testing a machine-learning model. It processes image data, potentially applies data augmentation (if enabled), resizes and normalizes it, and returns batches of images with their corresponding labels (**one-hot encoded**). The generator is useful for cases where large datasets need to be processed in small batches, on-the-fly, to avoid memory overload.



Let's break down the key components of the function:

def generator(source_path, folder_list, batch_size, is_train**=False**, augmention**=False**, debug**=False**):

- source_path: Base path where the images are stored.
- folder_list: List of folders (or subfolders) containing the images and their associated labels. Each entry contains the folder path and metadata separated by semicolons (e.g., foldername;imageType;label).
- batch_size: Number of data samples the generator will yield per iteration (usually for training or validation).
- is_train: A boolean flag indicating if the function is being used for training. If True, shuffling and augmentation may be applied.
- augmention: A boolean flag indicating whether to apply **data augmentation** techniques, such as image flipping, cropping, or resizing, to boost model generalization.
- debug: If True, it enables **debug visualization** using matplotlib to display the preprocessing and augmentation steps, useful for inspecting transformations.

Detailed Explanation of Key Steps:

---

1. Image Loading and Preprocessing

In each iteration of the generator, it processes images stored in folders (which act as units of data) found in folder_list. For each folder:

- Images are loaded using **OpenCV's cv2.imread** for further preprocessing.
- The image may be **cropped to maintain aspect ratio** if necessary.
- Resizing is done using cv2.resize to fit the output required dimensions (dim_y, dim_x).
- **Augmentation** may be applied (if enabled and if the random selection conditions are met):
  - **Edge Enhancement**
  - **Gaussian Blur**
  - **Detail Enhancement**
  - **Sharpening**
  - **Brightness Enhancement**

Note: This augmentation enhances robustness for model training by providing different "versions" of the images.

Here's an example of the augmentation decision and application code:

```
aug_type = None
if is_train and augmention and np.random.randint(0, 2) == 1:
    aug_type = np.random.randint(0, 4)


# Example for applying Brightness Enhancement
if aug_type == 4: # Brightness Enhancement augmention
    resized_im = np.array(ImageEnhance.Brightness(
        Image.fromarray(np.uint8(resized_im), 'RGB')).enhance(1.5))
```

---

2. Batch Data Construction

For each batch:

- **Batch data placeholders** are initialized — batch_data as a NumPy array of zeros in the shape of (batch_size, sequence_length, height, width, 3). The 3 represents the color channels (RGB).
- **Batch labels** are also initialized as a zero matrix of size (batch_size, num_classes) (where num_classes = 5 in this case) to store one-hot encoded label representations.

For every image inside the batch, the data and label are assigned conditionally:

- The code ensures **normalization** by scaling pixel values from the standard [0-255] range to the [0-1] range.

batch_data[folder, idx, :, :, 0] **=** resized_im[:, :, 0] **/** 255

batch_data[folder, idx, :, :, 1] **=** resized_im[:, :, 1] **/** 255

batch_data[folder, idx, :, :, 2] **=** resized_im[:, :, 2] **/** 255

---

3. One-Hot Encoding of Labels

Each folder path also contains its label in the string, accessible via:

label **=** int(folder_str.strip().split(';')[2])

For each folder, the corresponding **label is assigned as one-hot encoded**:

batch_labels[folder, label] **=** 1

If the dataset has 5 classes, the one-hot encoded label for class 2 would, for example, look like [0, 0, 1, 0, 0].

---

4. Handling Batches and Leftovers

- **Batch-wise iteration**: The generator uses a while True loop to continue yielding batches indefinitely, useful for model training.
- The folder_list is shuffled in two ways:
  - If is_train and augmention=True, it concatenates two permutations to effectively **double the dataset size** through augmenting images.
  - Otherwise, a **single permutation** is applied to shuffle during each epoch.

t **=** np.random.permutation(folder_list)

- **Handling the "leftover" data**: If the total dataset size is not perfectly divisible by the batch size, the last set of data (if smaller than batch_size) is handled and still yielded.

```
if leftover_count > 0:  # Handle leftover items
   ...
```

Advantages & Use Cases

1. **Efficient Memory Usage**: This generator optimizes memory usage by **loading only a batch of data** at a time rather than the entire dataset.
2. **On-the-fly Data Augmentation**: It provides real-time augmentation, creating additional training examples during the training process.
3. **Training & Testing Flexibility**: It can toggle both training (is_train=True) and testing (is_train=False) modes, enabling various behaviors accordingly.
4. **Debugging and Visualization**: If debug is enabled, the generator can display both the **original** and **augmented data**, helping you observe model input parameters.