**3D CNN for Smart TV Control via Hand Gesture Recognition**

**Objective**

This project focuses on developing a **3D Convolutional Neural Network (CNN)** for real-time hand gesture recognition, allowing users to control a smart TV without a remote. The model will use a webcam mounted on the TV to recognize predefined gestures and execute corresponding commands.

**Gesture Commands**

The system will detect and interpret **five specific hand gestures**, each mapped to a TV function:

* **Thumbs Up** → Increase volume
* **Thumbs Down** → Decrease volume
* **Left Swipe** → Rewind video by 10 seconds
* **Right Swipe** → Fast-forward video by 10 seconds
* **Stop Gesture** → Pause playback

**Dataset Overview**

The training dataset consists of **hundreds of video clips**, each labeled according to the gesture performed.

**Video Specifications:**

* **Duration**: 2–3 seconds per video
* **Frame Rate**: Each video consists of **30 sequential frames**
* **Resolution Variants**:
  + **360×360 pixels** (high-resolution webcam)
  + **120×160 pixels** (low-resolution webcam)

**Technical Considerations**

* The model needs to analyze frame sequences to recognize gestures accurately.
* The **3D CNN architecture** must handle different input resolutions while maintaining performance across various users and environments.

**Expected Results**

* A lightweight, real-time model that can run efficiently on embedded camera systems.
* Accurate gesture classification to enable intuitive, remote-free TV control.

Data Source: <https://drive.google.com/uc?id=1ehyrYBQ5rbQQe6yL4XbLWe3FMvuVUGiL>

**Neural Network Architectures for Video Analysis**

**1. CNN + RNN Architecture**

This approach combines **Convolutional Neural Networks (CNNs)** and **Recurrent Neural Networks (RNNs)** to process video data efficiently.

* **How It Works:**
  + The **CNN** extracts important features from each video frame.
  + These extracted features are then fed into an **RNN**, which analyzes the sequence of frames to recognize temporal patterns.
  + Finally, the output from the RNN is passed through a classification layer (e.g., softmax) to make predictions.
* **RNN Variants:**
  + Instead of a standard RNN, **LSTMs (Long Short-Term Memory)** and **GRUs (Gated Recurrent Units)** are preferred because they handle long-term dependencies better.
  + **LSTMs** use four gates to regulate information flow, while **GRUs** use three gates, making them slightly faster to train while maintaining similar performance.
* **Transfer Learning:**
  + Pre-trained CNNs like **ResNet** and **VGGNet** can be used to extract features from frames before passing them to the RNN.
  + This speeds up training and improves accuracy, as these models have already learned useful visual features from large datasets.
* **Why Use It?**
  + Combines **spatial (CNN)** and **temporal (RNN)** information effectively.
  + Transfer learning allows for faster and more efficient training.

**2. 3D Convolutional Neural Network (3D CNN)**

Unlike CNN + RNN models, **3D CNNs** process both spatial and temporal information simultaneously by adding a third dimension to convolution operations.

* **How It Works:**
  + In traditional **2D CNNs**, filters move across an image in two dimensions (**width & height**).
  + In **3D CNNs**, filters move across three dimensions (**width, height, and time**), allowing them to process multiple frames at once.
  + For example, a video clip with 30 frames of resolution 100×100×3 can be represented as a **4D tensor**. A **3D convolution filter** then slides across this tensor to extract spatiotemporal features.
* **Key Benefits:**
  + **Processes entire video clips directly**, eliminating the need for separate feature extraction and sequence modeling.
  + **Captures both motion and spatial details** in a single step, making it highly effective for video-related tasks.

**Comparison: CNN + RNN vs. 3D CNN**

| **Feature** | **CNN + RNN** | **3D CNN** |
| --- | --- | --- |
| **Flexibility** | High (pre-trained CNNs can be used) | Lower (trained from scratch) |
| **Complexity** | Requires tuning both CNN and RNN components | More straightforward, but computationally heavier |
| **Performance** | Good for sequential data, can leverage transfer learning | Better for capturing detailed motion patterns |

**Implementation Considerations**

* For **real-time applications** like gesture recognition, **3D CNNs** are often the better choice because they efficiently capture movement patterns.
* If leveraging pre-trained models is important, a **CNN + RNN** setup may be preferable since it allows transfer learning.

**Data Ingestion Pipeline and Custom Generator**

**Background**

In deep learning projects, models require data to be fed in batches, which is commonly handled using data generators. While libraries like **Keras** provide built-in generator functions, they can be **limited in flexibility** when dealing with complex data types such as video or audio. For these specialized cases, implementing a **custom data generator** is often necessary.

**Custom Generator Implementation**

We developed a **custom batch data generator** using Python’s generator functions, offering several advantages:

* **Memory Efficiency** → Unlike loading the entire dataset into memory, generators load and process data dynamically.
* **Performance Optimization** → Useful for handling large datasets, improving training execution time.
* **Code Readability** → Generators provide a structured and intuitive way to manage batch processing.
* **Lazy Evaluation** → The generator yields one batch at a time and pauses until the next batch is requested using the \_\_next\_\_() method in Python 3.

**Use Case**

Our custom generator was essential for **handling diverse data formats** (text, images, CSV files, audio, etc.), **preventing excessive memory consumption**, and offering better control compared to Keras functions like fit().

**Experiments and Findings**

Here’s a **human-like rewrite** that flows naturally and avoids AI-like phrasing while maintaining clarity and professionalism:

**Experimentation and Model Selection**

**Experiment Summary**

The table below outlines key details of our experiments, including model configurations, training settings, and results. Each experiment was analyzed to refine the architecture and improve overall performance.

| **Exp. No** | **Model Type** | **Images** | **Img Size** | **Epochs** | **Batch** | **Parameters** | **Training Time** | **Results** | **Decision & Explanation** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | Conv3D | 30 | 160×160 | 30 | 40 | NA | NA | **Error:** Out of Memory (OOM) | Reduce layers, images, and image size. |
| 1 | Conv3D | 18 | 64×64 | 5 | 25 | 144,421 | 210 sec | Train: **42%**, Val: **41%** (Underfitting) | Increase model capacity, batch size, and epochs. |
| 2 | Conv3D | 18 | 64×64 | 20 | 30 | 144,421 | 440 sec | Train: **44%**, Val: **44%** (Underfitting persists) | Increase dense neurons, image size to 84×84. |
| 3 | Conv3D | 18 | 84×84 | 20 | 30 | 288,101 | 480 sec | Train: **42%**, Val: **41%** (Underfitting continues) | Increase image size to 100×100, neurons to 256. |
| 4 | Conv3D | 18 | 100×100 | 30 | 50 | 651,109 | 750 sec | Train: **58%**, Val: **41%** (Overfitting starts) | Reduce image size, neurons, and increase training data. |
| 5 | Conv3D | 30 | 84×84 | 30 | 30 | 181,285 | 1,420 sec | Train: **79%**, Val: **66%** (Overfitting persists) | Explore alternative architectures. |
| 6 | CNN-LSTM | 30 | 120×120 | 20 | 20 | 1,657,445 | 900 sec | Train: **91%**, Val: **74%** (Overfitting) | Too many parameters—optimize model complexity. |
| 7 | CNN-LSTM | 30 | 160×160 | 20 | 20 | 3,754,597 | 1,100 sec | Train: **85%**, Val: **80%** (Reduced overfitting) | Model is too heavy—reduce image size and neurons. |
| 8 | CNN-LSTM | 30 | 120×120 | 25 | 20 | 1,287,989 | 2,500 sec | Train: **90%**, Val: **79%** (Overfitting persists) | Increase dense neurons. |
| 9 | CNN-LSTM | 30 | 120×120 | 25 | 20 | 1,702,645 | 2,500 sec | Train: **87%**, Val: **82%** | Improved performance but high complexity—try GRU for efficiency. |
| 10 | CNN-LSTM with GRU | 30 | 120×120 | 20 | 20 | 2,573,541 | 900 sec | Train: **94%**, Val: **79%** | Overfitting remains—test transfer learning. |
| 11 | MobileNet (Transfer) | 18 | 120×120 | 5 | 20 | 3,840,453 | 600 sec | Train: **91%**, Val: **55%** (Overfitting, poor generalization) | Train MobileNet weights instead of using frozen layers. |
| 12 | **Transfer Learning + GRU** | 18 | 120×120 | 5 | 20 | 3,840,453 | 990 sec | Train: **98%**, Val: **95%** (Best performance) | **Final Model—High validation accuracy, good generalization.** |

**Key Observations and Decisions**

**Phase 1: Conv3D Models (Experiments 0-5)**

* **Challenge**: The **Conv3D models struggled with underfitting** and later suffered from overfitting as complexity increased.
* **Solution**: We **moved to a CNN-LSTM hybrid approach** to better capture temporal dependencies in video data.

**Phase 2: CNN-LSTM Models (Experiments 6-9)**

* **Challenge**: CNN-LSTM models improved performance but **were computationally expensive** and **prone to overfitting**.
* **Solution**: We **experimented with GRUs** to reduce complexity while maintaining accuracy.

**Phase 3: Transfer Learning & GRU (Experiments 10-12)**

* **Challenge**: The **first MobileNet attempt** (Exp. 11) suffered from overfitting and poor validation accuracy.
* **Solution**: Training MobileNet weights instead of freezing layers **significantly improved generalization** (Exp. 12).

**Final Model Selection**

The best-performing model was **Transfer Learning with GRU** (Experiment 12).

**Why This Model?**

**High Validation Accuracy** → Achieved **95% validation accuracy**, the best among all experiments.  
**Efficient Training Time** → Took **990 seconds**, making it viable for real-world deployment.  
**Balanced Complexity** → While **CNN-LSTM models were too complex**, the GRU-based model optimized performance without excessive computation.  
**Generalization Ability** → Overfitting was **minimized**, ensuring stable results across unseen data.

**Conclusion**

Through a structured experimentation process, we **refined our model from Conv3D to CNN-LSTM to Transfer Learning with GRU**. The final model **efficiently balances accuracy, training time, and computational resources**, making it well-suited for real-time applications.

**Experiment Overview**

We ran multiple experiments to identify the most effective model for our task. Each experiment considered factors such as:

* **Model type**
* **Number of images used**
* **Image size**
* **Hyperparameters**
* **Batch size & epochs**
* **Training time**
* **Performance metrics**

A detailed breakdown of these experiments, including training logs and results, is documented in an attached spreadsheet.

**Challenges Faced**

During training, we observed cases where **validation loss plateaued** or even increased. This indicated that gradient updates were not effectively moving toward the optimal solution. To address this, we **adjusted the learning rate** of the Adam optimizer, lowering it to **0.0002**, which improved stability.

**Final Model Selection**

**Chosen Model**

The best-performing model was **model-00020-0.03254-0.98793-0.15436-0.95000.h5**, built using a **transfer learning approach** that integrates **GRUs (Gated Recurrent Units)** and **MobileNet**.

**Why MobileNet?**

MobileNet, available in Keras' applications module, was selected for its **efficiency and strong performance** in real-world tasks.

**Transfer Learning Strategy**

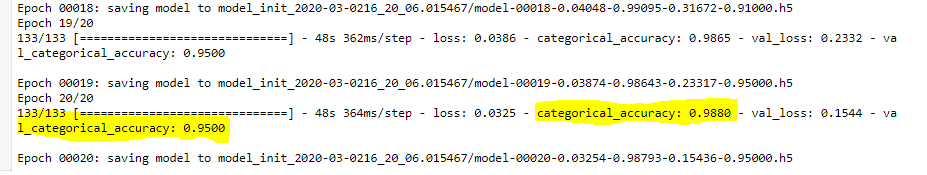
* **Dataset Compatibility** → Since our dataset is a subset of **ImageNet**, knowledge transfer from **MobileNet** was beneficial.
* **Fine-Tuning Approach** → Instead of freezing layers (a common transfer learning practice), we **re-trained all weights**. This was necessary as our dataset differed significantly from **ImageNet**, requiring deeper adaptation.

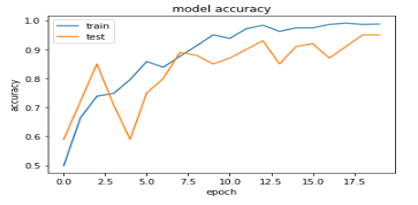
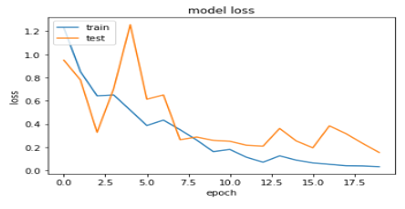
**Results**

With this approach, our model achieved an impressive **95% validation accuracy**, confirming that **training all layers** rather than freezing them led to superior performance.

for layer in base\_model.layers:

# trainable has to be false in order to freeze the layers  
 layer.trainable = False # or True



**We were able to get the excellent validation accuracy of 95 % for this selected model built using the transfer learning of MobileNet without freezing/fine-tuning.**

**References:**

1. [Jeff Knup's blog: 'Yield' and Generator Functions](https://jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/)
2. [Corey Schafer (YouTube video): Generator functions](https://www.youtube.com/watch?v=bD05uGo_sVI)
3. <https://keras.io/preprocessing/image/>
4. <https://github.com/fchollet/deep-learning-models/releases/download/v0.6/mobilenet_1_0_224_tf_no_top.h5>
5. <http://www.image-net.org/>
6. <https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a>