CHIROVOX

Thanks for the nudge—I'll give you both: a quick explanation of why we use Dense(128, activation='relu') and then a full, step-by-step breakdown of your CHIROVOX project: architecture, algorithms, libraries, training flow, and how each file fits in.

Why we use keras.layers.Dense(128, activation='relu')

- What it is: A fully connected layer with 128 neurons and ReLU nonlinearity.
- Where it sits: Right after Flatten(), just before the final softmax classifier in train.py.
- What it does:
    - Learns high-level, non-linear combinations of the features extracted by the Conv2D/MaxPool layers.
    - ReLU introduces non-linearity to model complex decision boundaries.
    - 128 units is a good balance for a small model (not too tiny, not too heavy) for 64×64 inputs.
- Effect on training:
    - Acts as a "feature combiner" before the final classification.
    - Helps the softmax layer see richer, more separable features.
- Rough math: For input feature vector x (from Flatten), Dense computes y = ReLU(Wx + b) with W ∈ R[F × 128], then the final Dense(num_classes, softmax) maps y to class probabilities.

Tip: If you see overfitting later, consider adding Dropout (e.g., Dropout(0.3)) before/after this Dense layer.

CHIROVOX: Full project overview

What it is

A sign-language helper that:

- Uses a CNN trained on gesture images
- Converts the model to TensorFlow.js
- Runs live camera detection in the browser

- Speaks the recognized sign in English/Sinhala/Tamil
- Includes learning (education) and dictionary pages

High-level architecture

- Training (Python, offline):
    - train.py trains a small CNN on images under pages/gesture/dataset/…
    - Exports a Keras model (gesture_model.h5) and labels.json
    - Converts to TFJS (writes model.json + shard files) under model
- Inference (browser, online/offline):
    - gesture.js loads the TFJS model
    - Captures camera frames (MediaPipe camera utils + <video>)
    - Optionally draws hands (MediaPipe Hands) and overlays
    - Resizes frames to 64×64 and feeds to the TFJS model
    - Speaks the predicted label using Web Speech Synthesis
    - Falls back to a finger-count heuristic if the model isn't available
- UI/UX:
    - index.html landing with header, language selector, theme toggle
    - education.html learning cards
    - dictionary.html table of common phrases
    - Shared theme in theme.css, plus page CSS
    - Global language state via lang.js synced across pages

Algorithms used

1) Image classification (CNN)

Defined in train.py:

- Input: 64×64 RGB images of hands (from folder structure)
- Data pipeline: ImageDataGenerator with rescaling and light augmentation
- Architecture:
    - Conv2D(32) + MaxPool
    - Conv2D(64) + MaxPool

    - Conv2D(128) + MaxPool
    - Flatten
    - Dense(128, relu) ← feature combiner
    - Dense(num_classes, softmax)
- Loss: categorical_crossentropy
- Optimizer: Adam
- Epochs: 10 (can increase if not overfitting)

2) TFJS conversion

- Uses tensorflowjs_converter to export TFJS artifacts for the web app.
- Client loads model.json and predicts per-frame.

3) Live detection pipeline

In gesture.js:

- Camera frames → canvas → resize to 64×64 → tf.browser.fromPixels → model.predict → argmax → label
- Optional fallback when TFJS model not found: heuristic finger counting via MediaPipe landmarks (maps finger count to a small set of gestures).
- TTS: Map predicted label to localized text and speak via SpeechSynthesis with language-specific voices.

Libraries used

Python (training)

- tensorflow==2.15
- tensorflowjs (for conversion)
- numpy
- matplotlib (optional plotting if added later)
- Standard library: os, json, subprocess, sys, pathlib

Browser (inference/UI)

- TensorFlow.js (CDN): @tensorflow/tfjs
- MediaPipe: hands, camera_utils, drawing_utils (CDN)
- Web Speech API: window.speechSynthesis (no extra lib)

- Vanilla JS/HTML/CSS

Project structure mapped to behavior

- train.py
    - Loads images from pages/gesture/dataset/{class_name}/...jpg
    - Augments, builds, trains CNN
    - Saves gesture_model.h5
    - Writes labels.json containing ordered class names
    - Converts model to TFJS into model (so browser can load)
- requirements.txt
    - Tooling to set up Python training environment
- gesture.html
    - Loads theme, TFJS, MediaPipe scripts
    - Displays orb, “Open” button, language select
    - Has <video id="camera"> and <canvas id="output"> for previews/overlays
- gesture.js
    - Model loading logic (tries multiple paths for model.json)
    - Inference on frames (64×64)
    - Rendering of landmarks (MediaPipe) and overlay drawing
    - Fallback heuristic if no model
    - Throttled TTS, language mapping, event sync
    - Theme/localStorage integration
- gesture.css
    - Camera area styling, buttons, layout, responsive tweaks
- lang.js
    - Global language store (localStorage)
    - Emits chirovox:langchange events so pages stay in sync
- index.html
    - Header with branding, language selector, theme toggle

    - Calls lang.js and wires theme/light-mode
    - Quick links to gesture/education/dictionary
- education.html
    - Title (“Learn Sri Lankan Sign Language”)
    - Language selector (synced)
    - Cards (image + heading + description); we enlarged images recently
- dictionary.html
    - Language selector (synced)
    - Glass-styled table of phrases
- theme.css and style.css
    - Global theme and page styles
    - Light/dark backgrounds (CHIROVOXWHITE.png for light mode)
    - Premium controls (gold language selector), layout polish
    - Education card grid; we updated image sizes
- quiz.html, schools.html, about.html
    - Additional content pages with shared theme and structure
    - Schools page includes a quick speak section
    - About includes project description and credits

Training: step-by-step how you did it (and how to re-run)

1. Prepare dataset

- Create class folders:
    - hello, iloveyou, yes, no, etc.
- Put sample images in each folder (JPG/PNG). More balanced data = better results.

2. Create/activate Python environment

- On Windows PowerShell:
    -
    -

-
-

3. Run training

- From project root:
  -
  -
  -
  -
- What it does:
  - Loads images with augmentation
  - Trains the CNN
  - Saves gesture_model.h5
  - Writes labels.json to labels.json
  - Converts to TFJS and writes model.json + shards to model

4. Start local server for testing

- From project root:
  -
  -
  -
  -
- Visit in browser:
  - Laptop: http://localhost:8000/pages/gesture/gesture.html
  - Phone (same Wi-Fi): http://<your-laptop-ip>:8000/pages/gesture/gesture.html
    - You may need HTTPS (use ngrok) for camera permissions on some devices

5. Use the camera page

- Click Open
- Show a known gesture

- You should see the class name and hear TTS in your chosen language

How predictions flow in the browser (detailed)

- gesture.js:
    - On Open:
        - Gets camera stream (MediaPipe Camera helper or getUserMedia fallback)
        - For each frame:
            - Draws the frame to canvas for preview
            - Converts frame to tensor:
                - tf.browser.fromPixels → resize to [64,64] → [0..1] normalization → [1, H, W, 3]
            - Runs model.predict(t) → argMax → predicted label
            - Displays label → maps to localized string → speaks with correct language voice (SpeechSynthesis)
    - If TFJS model fails to load:
        - Loads labels.json if available (for UI)
        - Uses a simple finger count heuristic to pick a plausible label

Improvements you can consider (optional)

- Add Dropout(0.3) before/after Dense(128) to reduce overfitting
- Add EarlyStopping and ModelCheckpoint callbacks
- Train longer (20–30 epochs) and monitor val accuracy
- Use transfer learning (e.g., MobileNetV2) for higher accuracy
- Use MediaPipe landmarks as input features for a small MLP (fast, low data)
- More consistent dataset lighting and orientations
- Balance class samples to avoid bias

Quick FAQ

- Why 64×64 input?
    - Keeps the model small for fast TFJS inference on phones. Increase if accuracy requires.

- Why we used ImageDataGenerator?
    - Simple augmentation improves generalization without adding code complexity.
- Why MediaPipe if we classify raw images?
    - MediaPipe helps visualize hand skeleton and provides a fallback heuristic. You can also switch to using landmark vectors as inputs.
- Why is HTTPS recommended on phone?
    - Browser camera APIs often require secure context (HTTPS) unless using localhost or some private LAN exceptions.