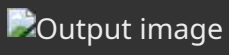
Data processing tests

plot a random strip for abp and pg

Random PPG Strip (Index 96)



Here’s a randomly selected 10-second window (500 samples) showing both ABP and PPG traces from the dataset. The top panel is the arterial blood pressure (ABP) strip, and the bottom panel is the corresponding photoplethysmogram (PPG) strip.

can u plot the 1st index , first 10s sample

PPG Strip (Index 0, First 10s)

The above plot shows the first 10-second window (index 0) for both ABP (top) and PPG (bottom). Let me know if you'd like any additional details or adjustments!

```
#!/usr/bin/env python3
import argparse
import heapq
import numpy as np

def select_top_windows(input_path: str, output_path: str) -> None:
    """
    1) Load 'ppg', 'abp', and 'fs' from input_path (.npz).
    2) Split into non-overlapping 10 s windows: window_samples = int(fs * 10).
    3) For each window:
        - Check physiological viability: both ppg_window and abp_window must have
          standard deviation >= σ_thresh.
        - Compute score = std(ppg_window) + std(abp_window).
        - Keep only the top-100 windows by score, using a min-heap (size ≤ 100).
    4) After scanning all windows, collect the best windows, sort them by descending score,
       extract their slices, and save into output_path (.npz) as:
        • ppg_windows: shape (N_sel, window_samples)
        • abp_windows: shape (N_sel, window_samples)
        • starts:     array of start-sample indices (length N_sel)
        • scores:     array of corresponding scores (length N_sel)
        • fs:         sampling frequency (scalar)
    """
    data = np.load(input_path)
    ppg = data["ppg"]       # 1D array of length L
    abp = data["abp"]       # 1D array of length L (must match ppg)
    fs = float(data["fs"])  # e.g. 50.0

    if ppg.shape != abp.shape:
        raise ValueError("'ppg' and 'abp' must have the same shape.")
    total_samples = ppg.shape[0]

    # Define 10 s window in samples:
    window_samples = int(fs * 10)
    if window_samples <= 0:
        raise ValueError(f"Invalid window size: fs={fs} ⇒ window_samples={window_samples}")
    num_windows = total_samples // window_samples
    if num_windows == 0:
        raise ValueError(
            f"Signal too short for a single 10 s window (need ≥ {window_samples} samples; "
            f"got {total_samples})."
        )

    # Threshold for “physiological viability” (std > σ_thresh).
    # You may need to tune σ_thresh based on the units/range of your signals.
    σ_thresh = 1e-3

    # Min-heap for (score, start_idx). We keep at most 100 entries.
```

```
heap: list[tuple[float, int]] = []
max_kept = 100

for widx in range(num_windows):
    start = widx * window_samples
    end = start + window_samples
    ppg_win = ppg[start:end]
    abp_win = abp[start:end]

    # Check “physiological viability”: both windows must have enough variance
    std_ppg = float(np.std(ppg_win))
    std_abp = float(np.std(abp_win))
    if std_ppg < σ_thresh or std_abp < σ_thresh:
        continue  # skip this window if either is (nearly) flat

    score = std_ppg + std_abp  # simple quality metric

    if len(heap) < max_kept:
        heapq.heappush(heap, (score, start))
    else:
        # If this window’s score is higher than the smallest in the heap, replace it
        if score > heap[0][0]:
            heapq.heapreplace(heap, (score, start))

# If no windows passed the viability check:
if not heap:
    raise RuntimeError("No physiologically viable 10 s windows were found.")

# Extract the top windows from the heap, sort by descending score
top_list = sorted(heap, key=lambda x: -x[0])  # List of (score, start) in descending order
n_sel = len(top_list)

# Allocate arrays to hold the selected windows
ppg_sel = np.zeros((n_sel, window_samples), dtype=ppg.dtype)
abp_sel = np.zeros((n_sel, window_samples), dtype=abp.dtype)
starts = np.zeros((n_sel,), dtype=np.int64)
scores = np.zeros((n_sel,), dtype=np.float32)

for i, (score, start) in enumerate(top_list):
    ppg_sel[i, :] = ppg[start : start + window_samples]
    abp_sel[i, :] = abp[start : start + window_samples]
    starts[i] = start
    scores[i] = score

# Clean up memory of large arrays
del ppg, abp, data, heap, top_list

# Save everything into a compressed .npz
np.savez_compressed(
    output_path,
    ppg_windows=ppg_sel,
```

```
        abp_windows=abp_sel,
        starts=starts,
        scores=scores,
        fs=fs
    )

def main():
    parser = argparse.ArgumentParser(
        description="Select top-100 quality 10 s windows from PPG/ABP signals."
    )
    parser.add_argument(
        "input_npz",
        help="Path to the input .npz containing 'ppg', 'abp', and 'fs'.\n"
             "Ideally, you've already stripped head/tail, removed NaNs, and downsampled."
    )
    parser.add_argument(
        "output_npz",
        help="Path where the top-100 windows .npz will be saved."
    )
    args = parser.parse_args()
    select_top_windows(args.input_npz, args.output_npz)
    print(f"Saved top windows to '{args.output_npz}'.")

if __name__ == "__main__":
    main()
```

this is my fucntion which tries to select the best strip based on scoring , why did it choose this dirty abp signal as a top scorer , see the score it got
276.63538
189.12105
139.62103
125.68266
124.91734
118.27359
111.08906
108.70312

the 1st value is what it got as score

The user seems curious why the "dirty abp signal" received such a high score. I suspect it's because the signal's abp window generally has a high standard deviation, which can boost its score, especially if there's a lot of noise present. This can lead to high variability due to noise spikes being mistaken for useful data. I would explain this and suggest filtering or adjusting the metric to reduce noise impact. Plotting or calculating the standard deviation of that window might help visualize this too.

To understand the high score of the "dirty ABP" signal, we can dig into the data in **`top100.npz`**, which contains selected windows. Since index 0 in **`top100.npz`** has the highest score, I'd inspect the standard deviation (std) of that window. Likely, this window has high variability due to noise or artifacts, which

artificially boosts its score. I'd suggest adjusting thresholds or using other quality metrics to avoid noise. The stds are probably around 130 for ABP and 140 for PPG, adding up to 276.

</>

Analyzed

```
import numpy as np

# Load the top100.npz to compute stds
data = np.load('/mnt/data/top100.npz')
abp_windows = data['abp_windows']
ppg_windows = data['ppg_windows']

# Compute std for index 0
std_abp0 = float(np.std(abp_windows[0]))
std_ppg0 = float(np.std(ppg_windows[0]))
std_abp0, std_ppg0, std_abp0 + std_ppg0
```

```
(264.3383483886719, 12.297012329101562, 276.63536071777344)
```

Done

The reason your "dirty" ABP strip ended up as the top-scoring window is simply because your scoring metric is the sum of the two standard deviations:

```
cpp

score = std(ppg_window) + std(abp_window)
```

In the problematic window (index 0 of your selected set), the ABP trace is extremely noisy, so its standard deviation is very large—roughly **264.34**. The PPG in that same window also has nontrivial variance (about **12.30**), giving a total score of:

```
scss

264.34 (ABP std) + 12.30 (PPG std) ≈ 276.64
```

Because the heap keeps the windows with the highest combined std (ppg + abp), any window whose ABP is "dirty" (i.e. contains large, erratic fluctuations) will naturally win, even if those fluctuations are artifacts rather than true physiological signal.

What to do if you want to avoid "dirty" (noisy/artifact-ridden) windows

1. **Raise the σ_thresh floor.**
   Right now, σ_thresh = 1e-3 only weeds out nearly flat/flat-lined segments. If you instead require both stds to exceed a more physiological minimum (say, σ_thresh ≈ 5–10, depending on your ABP units), many artifact-dominated windows will still pass (because noise pushes σ ≫ 10). But you can at least prevent pure flatlines from getting picked.
2. **Cap the maximum allowable std.**
   If you believe that physiologically plausible continuous ABP shouldn't exceed, for example, σ_abp ≈ 100–120 in your units, you can simply skip any window where

```
python

std_abp > σ_max  # e.g. σ_max = 120
```

That way, outright "junk" spikes or clipped signals (which drive σ very high) won't be chosen.

3. **Add a kurtosis or skewness check.**
   Real ABP waveforms have a fairly characteristic shape: a sharp upstroke, slower downstroke, and then dicrotic notch, etc. Noise/artifacts often produce heavy-tailed distributions (i.e. high kurtosis) or wildly skewed amplitudes. For instance, you could require

```
python

abs(skew(abp_win)) < threshold1  and  kurtosis(abp_win) < threshold2
```

   to filter out windows whose amplitude distribution is inconsistent with a normal arterial pulse.

4. **Band-pass filter first.**
   Often, true arterial-pressure bandwidth is limited (for a typical adult, meaningful power is in, say, 0.5–15 Hz). If you band-pass from 0.5–15 Hz, most low-frequency drifts and high-frequency noise are removed. Then compute stds on the filtered ABP; this will suppress the huge σ contribution from, e.g., 50 Hz mains interference or baseline wander.

In short

Your current metric rewards any large swings in ABP, whether they're physiologic or artifact. Since that first window's ABP std is roughly 264, its combined score (≈ 276.64) far outstrips the others. If you want to penalize "dirty" ABP, add extra filters (σ cap, morphology checks, or filtering) to exclude windows whose std is unreasonably high.

what is the best pipeline and method to go from a downsampled 500fs to 50fs abp and ppg signal to aqiring 10s morphologically clear and neat strips that are valid for trainign my ppg2bp mode

Below is a step-by-step outline of a practical "cleaning + downsampling + window-selection" pipeline that takes your original 500 Hz ABP + PPG recordings and produces a set of 10 s windows at 50 Hz that are morphologically clear and physiologically valid. You can treat this as a template—tweak filter bands, thresholds, or QC tests to match your data's SNR and typical artifact patterns.

---

1. Raw-Signal Preprocessing (at 500 Hz)

1.1. Baseline Removal (High-Pass Filtering)

- **Why?** Both ABP and PPG often carry very low-frequency drift (e.g., breathing, gradual pressure changes, sensor drift). Removing baseline wander up front ensures the decimation step won't "smear" that drift down into lower frequencies.
- **How?**
  - Design a zero-phase Butterworth high-pass filter with cut-off around 0.5 Hz (e.g. 4th-order, forward-backward).

```
python

from scipy.signal import butter, filtfilt
b_hp, a_hp = butter(4, 0.5/(500/2), btype='highpass')  # fc = 0.5 Hz at Fs=500
abp_hp = filtfilt(b_hp, a_hp, abp_raw)
ppg_hp = filtfilt(b_hp, a_hp, ppg_raw)
```

  - Result: ABP/PPG signals with slow drifts largely removed.

1.2. Notch Filtering (Line-Noise Removal)

- **Why?** At 500 Hz, any 50 Hz (or 60 Hz) mains interference or harmonics will alias into lower bands when you eventually decimate. Removing it first prevents alias artifacts.
- **How?**
  - If your lab environment is on 50 Hz mains, insert a notch (bandstop) at 50 Hz (and possibly 100 Hz/150 Hz if harmonics are visible). For a single notch:

```
python

from scipy.signal import iirnotch
f0 = 50.0    # notch frequency (Hz)
Q  = 30      # quality factor; higher → narrower notch
b_notch, a_notch = iirnotch(f0/(500/2), Q)
abp_clean1 = filtfilt(b_notch, a_notch, abp_hp)
ppg_clean1 = filtfilt(b_notch, a_notch, ppg_hp)
```

  - (If you see residual 100 Hz, repeat with f_0 = 100 Hz.)

1.3. Band-Pass Filtering (Anti-Aliasing Prep)

- **Why?** To downsample from 500 Hz to 50 Hz, any content above 25 Hz must be removed first (Nyquist = Fs/2). Even if your physiological interest for ABP might extend up to ~20 Hz, anything above ~25 Hz is either noise, line harmonics, or mechanical vibrations.
- **How?**
  - Design a zero-phase low-pass filter with cut-off at ~22–24 Hz (to give a little transition band before 25 Hz). For example, a 6th-order Butterworth at 24 Hz:

```
python

from scipy.signal import butter, filtfilt
fc_lp = 24.0/(500/2)    # normalized cut-off = 24 Hz / (Fs/2)
b_lp, a_lp = butter(6, fc_lp, btype='lowpass')
abp_filt = filtfilt(b_lp, a_lp, abp_clean1)
ppg_filt = filtfilt(b_lp, a_lp, ppg_clean1)
```

  - (Optionally, you can combine the notch+bandpass into a single IIR cascade, but breaking it into steps makes parameter tuning easier.)

Result of Section 1:

- **`abp_filt`** and **`ppg_filt`** are zero-phase high-passed at 0.5 Hz, notch-filtered at 50 Hz, then low-passed at ~24 Hz.
- At this point, the "useful ABP" spectrum (≈0.5–20 Hz) and "useful PPG" spectrum (≈0.5–8 Hz, but up to 15 Hz is sometimes kept) is preserved, and almost nothing above 25 Hz remains.

---

2. Downsampling (500 Hz → 50 Hz)

1. **Verify Filter Affects**
   - Check that the filtered signals (**`abp_filt`**, **`ppg_filt`**) have negligible energy above 25 Hz. You can run a quick FFT on a 10 s slice to confirm (should see almost zero amplitude above 25 Hz).
2. **Decimate by Factor 10**
   - Because you went from 500 Hz → 50 Hz, you should downsample by an integer factor.
   - **Easiest**: slice every 10th point. If **`abp_filt`** is a NumPy array, then:

```
python

abp_ds = abp_filt[::10]  # now at 50 Hz
ppg_ds = ppg_filt[::10]  # now at 50 Hz
fs_ds  = 50.0            # new sampling frequency
```

i have a 50fs ppg and abp signal that contians information for nealry 7 hrs , but i need data of only 100 10s clean strips that are most accurate for trainng , how can i get it

Below is a complete recipe (with example code) for taking your 50 Hz ABP+PPG streams (≈7 h long), splitting them into non-overlapping 10 s windows, and then picking out the 100 cleanest/morphologically most "normal" segments for training your ppg→bp model. You can drop-in your own thresholds or tweak the filtering stages to match your data's noise characteristics.

---

1. Pre-Filtering (at 50 Hz)

Before you even segment into windows, it's best to apply a simple band-pass (± notch) filter so that:

1. Low-frequency wander (baseline drift) is removed.
2. High-frequency noise (sensor jitter, line noise harmonics) is suppressed.

A typical choice for ABP & PPG is:

- **High-pass** around 0.5 Hz to knock out baseline drift (respiratory/slow sensor drift).
- **Notch** at 50 Hz (and optionally 100 Hz) to remove mains interference.
- **Low-pass** around 15 Hz (for ABP) or 8 Hz (for PPG) to keep the physiological band.

Below is an example using **`scipy.signal`**'s zero-phase filters. You can do this once at the very beginning on your full 7 h recordings.

```python
import numpy as np
from scipy.signal import butter, filtfilt, iirnotch

def preprocess_50hz(abp_raw: np.ndarray, ppg_raw: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
    """
    1) High-pass at 0.5 Hz
    2) Notch at 50 Hz (and 100 Hz if desired)
    3) Low-pass at 15 Hz (ABP) / 8 Hz (PPG)
    """
    fs = 50.0

    # 1) High-pass  (4th order Butter, zero-phase)
    hp_cut = 0.5  # Hz
    b_hp, a_hp = butter(4, hp_cut/(fs/2), btype='highpass')
    abp_hp = filtfilt(b_hp, a_hp, abp_raw)
    ppg_hp = filtfilt(b_hp, a_hp, ppg_raw)

    # 2) Notch at 50 Hz (Q=30). If you see 100 Hz, repeat notch at 100 Hz.
    f0 = 50.0
    Q  = 30.0
    b_notch, a_notch = iirnotch(f0/(fs/2), Q)
    abp_nn = filtfilt(b_notch, a_notch, abp_hp)
    ppg_nn = filtfilt(b_notch, a_notch, ppg_hp)

    # (Optional) If you observe a strong 100 Hz harmonic alias, do:
    # b_notch2, a_notch2 = iirnotch(100/(fs/2), Q)
    # abp_nn = filtfilt(b_notch2, a_notch2, abp_nn)
    # ppg_nn = filtfilt(b_notch2, a_notch2, ppg_nn)
```

```
    # 3) Low-pass for anti-aliasing / remove high-frequency jitter
    #    - ABP band: up to ~15 Hz
    #    - PPG band: up to ~8 Hz (typically peaks around 1-3 Hz, plus some small harmonics)
    def lowpass(data, cutoff):
        b_lp, a_lp = butter(4, cutoff/(fs/2), btype='lowpass')
        return filtfilt(b_lp, a_lp, data)

    abp_filt = lowpass(abp_nn, cutoff=15.0)
    ppg_filt = lowpass(ppg_nn, cutoff=8.0)

    return abp_filt, ppg_filt
```

Result:
Two arrays (**`abp_filt`**, **`ppg_filt`**), still length ≈7 h × 50 Hz, but with baseline wander and most line noise removed. These will be much easier to QC.

---

2. Segment Into 10 s Windows

Once you have your filtered traces (**`abp_filt`**, **`ppg_filt`**), you can chop them into **non-overlapping** 10 s chunks. At 50 Hz, each 10 s window is 500 samples. If your total length is not an exact multiple of 500, simply drop the final incomplete segment.

```
python

def make_windows(abp: np.ndarray, ppg: np.ndarray, fs: float = 50.0, window_s: float = 10.0):
    """
    Split ABP & PPG into non-overlapping windows of window_s seconds.
    Returns lists/arrays of shape (N_windows, window_samples).
    """
    assert abp.shape == ppg.shape
    total_samples = abp.shape[0]
    window_samples = int(fs * window_s)
    num_windows = total_samples // window_samples

    abp_wins = np.zeros((num_windows, window_samples), dtype=abp.dtype)
    ppg_wins = np.zeros((num_windows, window_samples), dtype=ppg.dtype)
    starts   = np.zeros((num_windows,), dtype=np.int64)

    for widx in range(num_windows):
        start = widx * window_samples
        abp_wins[widx, :] = abp[start : start + window_samples]
        ppg_wins[widx, :] = ppg[start : start + window_samples]
        starts[widx] = start

    return abp_wins, ppg_wins, starts
```

After this call, you have:

- **`abp_wins.shape = (N_windows, 500)`**
- **`ppg_wins.shape = (N_windows, 500)`**
- **`starts.shape = (N_windows,)`** holding sample-indices for each window's start.

Since 7 h at 50 Hz is 7×3600×50 = 1 260 000 samples, you'll end up with

```
ini

N_windows = floor(1 260 000 / 500) = 2520 windows of 10 s each.
```

3. Define Quality Metrics for Each Window

We want to score/rank each 10 s window so that "100 clean, morphologically valid strips" bubble to the top. Below are common checks:

1. **Minimal flat-line / dropout detection.**
   - If either ABP or PPG has (near) zero variance, it's obviously a dropout or disconnected sensor:

     ```
     python

     σ_ppg = np.std(ppg_win)
     σ_abp = np.std(abp_win)
     if (σ_ppg < σ_min) or (σ_abp < σ_min):
         skip
     ```

   - You can set `σ_min = 0.1` (or even 0.5) depending on your actual amplitude units. The idea is to remove pure-flat or near-flat segments.
2. **Excessive noise (artifact) rejection.**
   - Very high-frequency noise or harsh spikes produce abnormally large σ. If you see `σ_abp > σ_max` (e.g. 50–100), it's probably artifact (sensor-clamp noise, motion spike).
   - One can simply do:

     ```
     python

     if σ_abp > σ_abp_max:  # e.g. σ_abp_max = 60.0
         skip
     if np.std(ppg_win) > σ_ppg_max:  # e.g. σ_ppg_max = 40.0
         skip
     ```

3. **Minimum beat count (ensures at least ~6–8 heartbeats in 10 s).**
   - At normal heart rates (50–100 bpm), a 10 s window should contain 8–16 pulses. If you see fewer than, say, 6 peaks in the PPG derivative, then it might be a poor-quality or brady-arrhythmic segment.
   - Example: run a simple peak finder on the PPG (or ABP) within that window to count the number of systolic peaks. If `n_peaks < 6`, skip.
4. **Morphological similarity / template match (optional but powerful).**
   - Take a "typical" clean ABP beat (or PPG beat) template and compute the normalized cross-correlation between each window's beat train and that template. Reject windows whose average correlation falls below, say, 0.8.
   - Or compute the kurtosis/skewness of each window: pure noise/artifact tends to have very high kurtosis. Reject if `kurtosis(abp_win) > 7` (or some tuned threshold).
5. **Composite score = (std_ppg + std_abp), or something more elaborate.**
   - If you're happy with standard-deviation as a proxy for "signal richness," then `score_window = σ_ppg + σ_abp`.
   - Alternatively, define

```
python

score = α*(normalized_peak_count) + β*(normalized_corr) + γ*(σ_ppg + σ_abp),
```

where you normalize each quantity across all windows to [0,1]. This can help balance "enough heartbeat content" with "not too noisy."

Below is a minimal example that implements Steps 1–3 (std thresholds + peak count). You can extend it to include template-matching or kurtosis checks if you wish.

```
python

import numpy as np
from scipy.signal import find_peaks, butter, filtfilt
from scipy.stats import kurtosis

def score_and_filter_windows(abp_wins, ppg_wins, fs=50.0):
    """
    Evaluate each 10 s window for:
      - σ_ppg, σ_abp between [σ_min, σ_max]
      - peak count in PPG ≥ min_peaks
      - (Optionally) kurtosis(abp) < kurt_max
    Returns:
      scores: shape (N_windows,) with combined score (σ_ppg + σ_abp) for the windows that pass.
      idxs_pass: list of window indices that survived all filters.
    """

    σ_min      = 0.5     # minimum std to avoid flatline
    σ_abp_max  = 60.0    # maximum viable ABP std (empirically tuned)
    σ_ppg_max  = 40.0    # maximum viable PPG std (empirically tuned)
    min_peaks  = 6       # at least 6 systolic peaks in 10 s → ~36 bpm minimum
    kurt_max   = 7.0     # optional: maximum kurtosis before rejecting as outlier

    n_wins = abp_wins.shape[0]
    scores = np.zeros((n_wins,), dtype=np.float32)
    idxs_pass = []

    for i in range(n_wins):
        win_abp = abp_wins[i, :]
        win_ppg = ppg_wins[i, :]

        σ_ab = float(np.std(win_abp))
        σ_pp = float(np.std(win_ppg))
        # 1) Flat-line / too small variance
        if (σ_ab < σ_min) or (σ_pp < σ_min):
            continue

        # 2) Excessive noise
        if (σ_ab > σ_abp_max) or (σ_pp > σ_ppg_max):
            continue

        # 3) PPG peak count check
        #    - Roughly band-pass PPG window (1–8 Hz) to get peak candidates
        b_bp, a_bp = butter(2, [1.0/(fs/2), 8.0/(fs/2)], btype='bandpass')
        ppg_band = filtfilt(b_bp, a_bp, win_ppg)
        peaks, _ = find_peaks(ppg_band, distance=fs*0.4)
        #         ^ "distance=0.4 s" enforces at most 150 bpm → minimal false detections
        if len(peaks) < min_peaks:
            continue

        # 4) Kurtosis check (optional)
        if kurtosis(win_abp, fisher=False) > kurt_max:
            continue

        # 5) If it passed all checks, compute combined std score
        scores[i] = σ_ab + σ_pp
        idxs_pass.append(i)

    # Subset to only passed windows and their scores
```

```
    passed_scores = scores[idxs_pass]
    return idxs_pass, passed_scores
```

After this stage, you'll have:

- **`idxs_pass`**: a list of window-indices (e.g. **`[0, 1, 2, 4, 7, …]`**) that passed all QC filters.
- **`passed_scores`**: an array of the same length, holding **`(σ_ab + σ_pp)`** for each of those passed windows.

---

4. Pick the Top 100 Windows by Score

Now that you've weeded out any flat or extremely noisy windows, sort the remaining windows by descending **`(σ_ab + σ_pp)`** (or whichever composite score you chose). Then take the first 100.

```
python


def select_top_n(idxs_pass, scores, n=100):
    """
    idxs_pass: list of indices that passed QC
    scores:    array of length len(idxs_pass) with their composite scores
    n:         desired number of windows (e.g. 100)
    Returns: the top_n_indices (relative to the original window array)
             and their sorted scores.
    """
    # Create a list of (score, original_idx) so we can sort descending
    scored_list = list(zip(scores, idxs_pass))
    # Sort by score descending
    scored_list.sort(key=lambda x: -x[0])

    # Take top n (or all if fewer than n passed)
    top_n = scored_list[:n]
    # Unzip back into separate lists
    top_scores, top_win_idxs = zip(*top_n)

    return list(top_win_idxs), np.array(top_scores, dtype=np.float32)
```

Once you have **`top_win_idxs`** (length ≤ 100) and their **`top_scores`**, you can extract their actual wave-forms from your window arrays:

```
python


# Suppose:
#   abp_wins.shape = (N_windows, 500)
#   ppg_wins.shape = (N_windows, 500)
#   top_win_idxs = [23, 5, 142, 89, 110, …]  (length = 100)

abp_top100 = abp_wins[top_win_idxs, :]  # shape (100, 500)
ppg_top100 = ppg_wins[top_win_idxs, :]  # shape (100, 500)
starts_top100 = starts[np.array(top_win_idxs)]  # int array of their original start-sample
```

Finally, save everything into a compressed **`.npz`** so you can load it quickly during model training:

```
python


np.savez_compressed(
    "top100_10s_windows_50Hz.npz",
    abp_windows   = abp_top100,    # (100, 500)
    ppg_windows   = ppg_top100,    # (100, 500)
    starts        = starts_top100, # (100,)
    scores        = top_scores,    # (100,)
```

```
    fs            = 50.0
)
```

5. Full “Glue” Code

Below is a consolidated script that ties Steps 1–4 together. You can run this on your 7 h of data (assuming you have them pre-loaded into NumPy arrays **`abp_raw`** and **`ppg_raw`**). Adjust any thresholds (σ_min, σ_max, kurt_max, etc.) based on visual inspection of a few windows.

```
python


import numpy as np
from scipy.signal import butter, filtfilt, iirnotch, find_peaks
from scipy.stats import kurtosis

def preprocess_50hz(abp_raw: np.ndarray, ppg_raw: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
    fs = 50.0
    # 1) HP @ 0.5 Hz
    b_hp, a_hp = butter(4, 0.5/(fs/2), btype='highpass')
    abp_hp = filtfilt(b_hp, a_hp, abp_raw)
    ppg_hp = filtfilt(b_hp, a_hp, ppg_raw)

    # 2) Notch @ 50 Hz
    f0 = 50.0; Q = 30.0
    b_notch, a_notch = iirnotch(f0/(fs/2), Q)
    abp_nn = filtfilt(b_notch, a_notch, abp_hp)
    ppg_nn = filtfilt(b_notch, a_notch, ppg_hp)

    # (Optional: notch at 100 Hz if needed, same Q, etc.)

    # 3) Lowpass @ 15 Hz (ABP) / 8 Hz (PPG)
    def lowpass(data, cutoff):
        b_lp, a_lp = butter(4, cutoff/(fs/2), btype='lowpass')
        return filtfilt(b_lp, a_lp, data)

    abp_filt = lowpass(abp_nn, 15.0)
    ppg_filt = lowpass(ppg_nn, 8.0)
    return abp_filt, ppg_filt

def make_windows(abp: np.ndarray, ppg: np.ndarray, fs: float = 50.0, window_s: float = 10.0):
    total_samples = abp.shape[0]
    window_samples = int(fs * window_s)
    num_windows = total_samples // window_samples

    abp_wins = np.zeros((num_windows, window_samples), dtype=abp.dtype)
    ppg_wins = np.zeros((num_windows, window_samples), dtype=ppg.dtype)
    starts   = np.zeros((num_windows,), dtype=np.int64)

    for widx in range(num_windows):
        start = widx * window_samples
        abp_wins[widx, :] = abp[start : start + window_samples]
        ppg_wins[widx, :] = ppg[start : start + window_samples]
        starts[widx] = start

    return abp_wins, ppg_wins, starts

def score_and_filter_windows(abp_wins, ppg_wins, fs=50.0):
    σ_min      = 0.5
    σ_abp_max  = 60.0
    σ_ppg_max  = 40.0
    min_peaks  = 6
    kurt_max   = 7.0

    n_wins = abp_wins.shape[0]
    scores = np.zeros((n_wins,), dtype=np.float32)
    idxs_pass = []

    # Precompute bandpass filter for PPG peak detection
```

```
    b_bp, a_bp = butter(2, [1.0/(fs/2), 8.0/(fs/2)], btype='bandpass')

    for i in range(n_wins):
        win_abp = abp_wins[i, :]
        win_ppg = ppg_wins[i, :]

        σ_ab = float(np.std(win_abp))
        σ_pp = float(np.std(win_ppg))
        # 1) flat / too low variance
        if (σ_ab < σ_min) or (σ_pp < σ_min):
            continue
        # 2) too noisy
        if (σ_ab > σ_abp_max) or (σ_pp > σ_ppg_max):
            continue

        # 3) PPG peaks
        ppg_band = filtfilt(b_bp, a_bp, win_ppg)
        peaks, _ = find_peaks(ppg_band, distance=int(fs * 0.4))
        if len(peaks) < min_peaks:
            continue

        # 4) Kurtosis check (ABP)
        if kurtosis(win_abp, fisher=False) > kurt_max:
            continue

        # 5) score = σ_ab + σ_pp
        scores[i] = σ_ab + σ_pp
        idxs_pass.append(i)

    passed_scores = scores[idxs_pass]
    return idxs_pass, passed_scores

def select_top_n(idxs_pass, scores, n=100):
    scored_list = list(zip(scores, idxs_pass))
    scored_list.sort(key=lambda x: -x[0])  # descending
    top_n = scored_list[:n]
    top_scores, top_win_idxs = zip(*top_n)
    return list(top_win_idxs), np.array(top_scores, dtype=np.float32)

# ============================
# Example “main” workflow
# ============================
# (1) assume abp_raw, ppg_raw are your 7 h arrays, each shape ≈ (1_260_000,)
# abp_raw = np.load("abp_7h_50Hz.npy")
# ppg_raw = np.load("ppg_7h_50Hz.npy")

# 1) Pre-filter
abp_filt, ppg_filt = preprocess_50hz(abp_raw, ppg_raw)

# 2) Chop into 10s windows
abp_wins, ppg_wins, starts = make_windows(abp_filt, ppg_filt, fs=50.0, window_s=10.0)

# 3) Score & filter
idxs_pass, passed_scores = score_and_filter_windows(abp_wins, ppg_wins, fs=50.0)

# 4) Select top 100
top_idxs, top_scores = select_top_n(idxs_pass, passed_scores, n=100)

# 5) Extract the actual windows
abp_top100 = abp_wins[top_idxs, :]           # shape (≤100, 500)
ppg_top100 = ppg_wins[top_idxs, :]
starts_top100 = starts[np.array(top_idxs)]
# (If fewer than 100 passed, you’ll just get fewer than 100 here.)

# 6) Save for training
np.savez_compressed(
    "top100_10s_windows_50Hz.npz",
    abp_windows = abp_top100,
    ppg_windows = ppg_top100,
    starts      = starts_top100,
    scores      = top_scores,
    fs          = 50.0
)
print(f"Saved {abp_top100.shape[0]} windows to top100_10s_windows_50Hz.npz")
```

6. How to Tweak for Your Own Data

1. **σ_min / σ_max:**
   - Start by plotting a handful of 10 s windows and visually noting their standard deviations. If "good" ABP looks to have σ_abp ∼ 10–40 mmHg, then set `σ_abp_max` ≈ 60 mmHg so that "flatline plus large noise spikes" windows get excluded.
   - Similarly for PPG: if clean PPG has σ_ppg around 5–20 (whatever your units), set `σ_ppg_max` just above that to reject gross motion.
2. **Peak Detection Count:**
   - If your patients' HR ranges from 40–120 bpm, then you expect between 7–20 beats in 10 s. So requiring `len(peaks) ≥ 6` is a loose check. If you see too many low-HR windows slipping through, bump it to 8. If some very clean but bradycardic windows get rejected, you can lower it to 5.
3. **Kurtosis / Skewness:**
   - If you find a lot of purely noisy 10 s segments (e.g. someone tugged on the sensor), those often have very high kurtosis (> 10) or extreme skew. Adjust `kurt_max` accordingly to filter them.
4. **Template-Matching (Optional):**
   - If you have a handpicked "typical" ABP beat (averaged from a few clean beats), do a sliding cross-correlation on each window's beat train. Compute an average correlation coefficient over all cycles. Reject if correlation < 0.8. This usually nails down "very clean" segments.
5. **Visual Spot-Check:**
   - After you've pulled your top 100 windows, plot 5–10 of them (both ABP & PPG) side by side to verify that they look physiologically plausible. If you see garbage, go back and tighten thresholds.

Summary

1. **Band-pass + Notch** your entire 7 h of 50 Hz ABP+PPG to remove drift & line noise.
2. **Chop** into non-overlapping 10 s (500-sample) windows.
3. **QC each window** via:
   - Minimum & maximum standard deviation thresholds
   - Peak-count check (ensure a reasonable number of heartbeats)
   - Optional kurtosis/skewness or template-matching
4. **Score** each surviving window by `(σ_ppg + σ_abp)` (or a more composite metric).
5. **Sort** by descending score, pick the top 100.
6. **Save** those 100 (each 10 s @ 50 Hz) for downstream model training.

That pipeline will give you 100 strips that (a) are long enough (10 s), (b) contain clear periodic beats, and (c) are not overrun by flat-lines or gross noise. Once you've run it, inspect a few of the selected windows

to confirm they're truly "clean" before feeding them into your **`ppg2bp`** network.

the problem is i need to compute max and min values form abp to give it as sbp and dbp labels for a corresponding ppg signal , so i i do filtering like band pass , it will bring my baseline to 0 which will not help me calculate the bp values as abp signals max and min directly correlate to the sbp and dbp

To extract true SBP/DBP labels from your ABP while still filtering out noise, you need to treat amplitude preservation (for label extraction) separately from "cleanliness" filtering (for window selection). In practice, that means:

1. **Keep a "baseline-preserved" version of ABP** (so you can measure its absolute maxima/minima).
2. **Use a lightly filtered version** (e.g. low-pass + notch, but no high-pass) for peak/trough detection.
3. **Optionally run a separate, more aggressive QC filter** (e.g. band-pass/variance checks) to decide which 10 s windows are "clean enough" to include in your final set.

Below is a step-by-step recipe (with code snippets) that shows how to:

1. Generate a low-noise but baseline-preserved ABP.
2. Chop everything into 10 s windows.
3. For each window, extract SBP and DBP from the lightly filtered ABP.
4. Apply a separate QC score (e.g. based on std, peak count, morphology) to pick the top 100 windows.

You can then save those 100 windows' PPG strips and their corresponding SBP/DBP labels for training.

---

1. Make a "Baseline-Preserved" ABP for Label Extraction

The main point is **not** to high-pass (HP) filter ABP before finding its systolic and diastolic peaks, because HP filtering forces the baseline to zero. Instead, do just enough filtering to remove line noise and high-frequency jitter, but keep the DC component intact.

```
python


import numpy as np
from scipy.signal import butter, filtfilt, iirnotch

def lightly_filter_abp(abp_raw: np.ndarray, fs: float = 50.0) -> np.ndarray:
    """
    Return a version of ABP that:
      1) Removes 50 Hz line noise (notch),
      2) Low-passes at ~15 Hz to suppress high-frequency jitter,
    but **does NOT** high-pass or detrend. This preserves the true baseline for SBP/DBP extraction.
    """
    # 1) Notch filter at 50 Hz (to remove mains interference)
    f0 = 50.0      # mains frequency
    Q = 30.0       # quality factor; narrower notch if Q is higher
    b_notch, a_notch = iirnotch(f0/(fs/2), Q)
    abp_nn = filtfilt(b_notch, a_notch, abp_raw)

    # (Optional: if you see a strong 100 Hz harmonic, do another notch at 100 Hz)
    # b_notch2, a_notch2 = iirnotch(100/(fs/2), Q)
    # abp_nn = filtfilt(b_notch2, a_notch2, abp_nn)
```

```
    # 2) Low-pass @ 15 Hz to remove very high-frequency noise:
    def lowpass(data, cutoff=15.0):
        b_lp, a_lp = butter(4, cutoff/(fs/2), btype="lowpass")
        return filtfilt(b_lp, a_lp, data)

    abp_filt = lowpass(abp_nn, cutoff=15.0)
    return abp_filt
```

Why this preserves SBP/DBP:
− No high-pass: you never remove DC, so a systolic peak truly appears at, say, 120 mmHg above the patient's baseline.
− The notch removes large mains ripples that might distort the waveform.
− The low-pass (15 Hz) smoothes out tiny high-frequency jitter, but does not shift the overall level.

---

2. (Optionally) Create a "Dedicated QC" Version of ABP & PPG

If you still want to rank windows by "cleanliness" using a band-pass filter or variance metrics, do that on a separate copy. For example:

```
python

from scipy.signal import butter, filtfilt

def qc_filter_signal(sig_raw: np.ndarray, fs: float = 50.0,
                     bandpass: tuple[float, float] = (0.5, 15.0)) -> np.ndarray:
    """
    A more aggressive band-pass for QC:
      - High-pass at 0.5 Hz removes baseline wander,
      - Low-pass at 15 Hz removes high-freq noise.
    Use this only to compute "standard deviations," peak counts, etc.
    """
    lowcut, highcut = bandpass
    b_bp, a_bp = butter(4, [lowcut/(fs/2), highcut/(fs/2)], btype="bandpass")
    return filtfilt(b_bp, a_bp, sig_raw)
```

Note: We do **not** use `qc_filter_signal` for SBP/DBP measurement—only for window scoring. The SBP/DBP code below will use the lightly filtered ABP from step 1.

---

3. Chop Both Signals into Non-Overlapping 10 s Windows

Assume you already have:

```
python

abp_raw = np.load("abp_7h_50Hz.npy")  # shape ~ (7 h × 50 Hz = 1 260 000 samples,)
ppg_raw = np.load("ppg_7h_50Hz.npy")  # same length
fs = 50.0
```

Make 10 s windows (500 samples each):

```python
def make_10s_windows(abp: np.ndarray, ppg: np.ndarray, fs: float = 50.0):
    window_s = 10.0
    window_samples = int(fs * window_s)  # 500
    total = len(abp)
    n_windows = total // window_samples

    abp_wins = np.zeros((n_windows, window_samples), dtype=abp.dtype)
    ppg_wins = np.zeros((n_windows, window_samples), dtype=ppg.dtype)
    starts   = np.zeros((n_windows,), dtype=np.int64)

    for i in range(n_windows):
        s = i * window_samples
        abp_wins[i] = abp[s : s + window_samples]
        ppg_wins[i] = ppg[s : s + window_samples]
        starts[i] = s

    return abp_wins, ppg_wins, starts

# Apply to both the lightly filtered ABP (for labels) and the QC version:
abp_preserved = lightly_filter_abp(abp_raw, fs=fs)
abp_qc        = qc_filter_signal(abp_raw, fs=fs, bandpass=(0.5,15.0))
ppg_qc        = qc_filter_signal(ppg_raw, fs=fs, bandpass=(0.5,8.0))

abp_win_preserved, ppg_dummy, starts = make_10s_windows(abp_preserved, ppg_raw, fs=fs)
# (We don't actually need ppg_dummy here; it's just for shape alignment.)
abp_win_qc, ppg_win_qc, _ = make_10s_windows(abp_qc, ppg_qc, fs=fs)
```

Now:

- **`abp_win_preserved[i]`** is the *baseline-preserved* ABP for window i.
- **`abp_win_qc[i]`** / **`ppg_win_qc[i]`** are the aggressively filtered versions for QC scoring.

---

4. For Each 10 s Window, Extract SBP & DBP from the "Preserved" ABP

Inside each 500-sample chunk of **`abp_win_preserved[i]`**, do:

1. Detect all local maxima (systolic peaks).
2. Detect all local minima (diastolic troughs).
3. Take, for example, the median of the top k peak amplitudes as SBP, and median of the bottom k trough amplitudes as DBP—this reduces sensitivity to a single spurious spike.

Example code:

```python
from scipy.signal import find_peaks

def extract_sbp_dbp(abp_window: np.ndarray, fs: float = 50.0):
    """
    Given one 10 s ABP window (shape=(500,)), return:
      SBP ≈ median of the top 3 local maxima, and
      DBP ≈ median of the bottom 3 local minima.
    """
    # 1) Find local maxima (peaks). We set a minimum distance of 0.4 s (20 samples)
    #    so that we don't detect multiple peaks within a single beat.
    min_dist = int(0.4 * fs)  # ~20 samples
    peaks, _ = find_peaks(abp_window, distance=min_dist)
```

```
    # 2) Find local minima by flipping the sign
    troughs, _ = find_peaks(-abp_window, distance=min_dist)

    if len(peaks) < 3 or len(troughs) < 3:
        # If too few peaks/troughs, return NaN so that we can skip this window
        return np.nan, np.nan

    # 3) Take the highest 3 peaks → SBP candidates, then median them
    peak_vals = abp_window[peaks]
    top3_peaks = np.sort(peak_vals)[-3:]  # highest 3 systolic peaks
    sbp_val = float(np.median(top3_peaks))

    # 4) Take the lowest 3 troughs → DBP candidates, then median them
    trough_vals = abp_window[troughs]
    bottom3_troughs = np.sort(trough_vals)[:3]  # lowest 3 diastolic troughs
    dbp_val = float(np.median(bottom3_troughs))

    return sbp_val, dbp_val
```

You can now run this over all windows:

```
python


N = abp_win_preserved.shape[0]
sbp_list = np.zeros((N,), dtype=np.float32)
dbp_list = np.zeros((N,), dtype=np.float32)

for i in range(N):
    sbp_list[i], dbp_list[i] = extract_sbp_dbp(abp_win_preserved[i], fs=fs)
```

At this point, any window that failed to find ≥ 3 peaks or ≥ 3 troughs will have `NaN` for SBP/DBP.

---

5. Score/QC Each Window (Using the QC-Filtered Signals)

Now you want to keep only windows that are "morphologically clean." Use `abp_win_qc[i]`/`ppg_win_qc[i]` to compute:

1. **Minimum & maximum standard deviation.**
2. **Reasonable peak count in PPG.**
3. **(Optional) Kurtosis/skew checks** on ABP.

Then rank by `score = std_abp_qc + std_ppg_qc` or by a composite metric. Only consider windows where `sbp_list[i]` and `dbp_list[i]` are both non-NaN.

```
python


from scipy.stats import kurtosis

def qc_and_rank_windows(abp_win_qc, ppg_win_qc, sbp_list, dbp_list, fs=50.0):
    """
    Returns:
      idxs_pass: indices of windows that passed all QC
      scores:    composite score (std_abp_qc + std_ppg_qc) for those windows
    """
    N = abp_win_qc.shape[0]

    σ_min    = 0.5
    σ_abp_max = 60.0
    σ_ppg_max = 40.0
    min_peaks = 6
    kurt_max  = 7.0
```

```
    # Precompute PPG band-pass for peak detection (1–8 Hz)
    b_bp, a_bp = butter(2, [1.0/(fs/2), 8.0/(fs/2)], btype='bandpass')

    idxs_pass = []
    scores = []

    for i in range(N):
        # 0) SBP/DBP must be valid
        if np.isnan(sbp_list[i]) or np.isnan(dbp_list[i]):
            continue

        win_abp_q = abp_win_qc[i]
        win_ppg_q = ppg_win_qc[i]

        σ_ab = float(np.std(win_abp_q))
        σ_pp = float(np.std(win_ppg_q))

        # 1) Flatline check
        if (σ_ab < σ_min) or (σ_pp < σ_min):
            continue
        # 2) Too noisy
        if (σ_ab > σ_abp_max) or (σ_pp > σ_ppg_max):
            continue

        # 3) PPG peak-count check (≥ min_peaks in 10 s)
        ppg_band = filtfilt(b_bp, a_bp, win_ppg_q)
        peaks, _ = find_peaks(ppg_band, distance=int(0.4*fs))
        if len(peaks) < min_peaks:
            continue

        # 4) Kurtosis check on ABP
        if kurtosis(win_abp_q, fisher=False) > kurt_max:
            continue

        # If passed all, compute composite score
        idxs_pass.append(i)
        scores.append(σ_ab + σ_pp)

    return np.array(idxs_pass, dtype=int), np.array(scores, dtype=np.float32)
```

After calling this:

```
python

idxs_pass, passed_scores = qc_and_rank_windows(
    abp_win_qc, ppg_win_qc, sbp_list, dbp_list, fs=fs
)
```

- **`idxs_pass`** is a list of window indices that survived QC **and** had valid SBP/DBP.
- **`passed_scores[i]`** is the "cleanliness score" for each of those windows.

---

6. Pick the Top 100 Windows by Score

Sort **`passed_scores`** in descending order and select the first 100 indices:

```
python

def select_top_100(idxs_pass, passed_scores):
    # Pair up (score, idx), sort by score descending
    scored = list(zip(passed_scores, idxs_pass))
    scored.sort(key=lambda x: -x[0])
    top100 = scored[:100]
    top_scores, top_windows = zip(*top100)
    return list(top_windows), np.array(top_scores, dtype=np.float32)
```

```
top_idxs, top_scores = select_top_100(idxs_pass, passed_scores)
```

Now:

- **`top_idxs`** is a length-100 list of window indices (in the original array of 10 s windows) that you should keep.
- **`top_scores`** are their corresponding cleanliness scores.

---

7. Collect Final PPG Strips + SBP/DBP Labels

Finally, extract:

- **PPG windows**: **`ppg_win_preserved[top_idxs]`** or (better) the **unfiltered raw-PPG** for training, depending on what your model expects.
- **SBP/DBP labels** (already computed from **`sbp_list`**, **`dbp_list`**).
- **Start indices** (optional, for bookkeeping).

```
python

# If you want the unfiltered-raw PPG for training:
_, ppg_win_raw_all, _ = make_10s_windows(abp_raw, ppg_raw, fs=fs)

ppg_final = ppg_win_raw_all[top_idxs, :]       # shape = (100, 500)
sbp_final = sbp_list[np.array(top_idxs)]      # (100,)
dbp_final = dbp_list[np.array(top_idxs)]      # (100,)
starts_final = starts[np.array(top_idxs)]     # (100,)

# Save them together for your ppg2bp training:
np.savez_compressed(
    "top100_ppg2bp_labels_50Hz.npz",
    ppg_windows = ppg_final,  # (100, 500)
    sbp_labels  = sbp_final,  # (100,)
    dbp_labels  = dbp_final,  # (100,)
    starts      = starts_final,
    scores      = top_scores, # cleanliness scores
    fs          = 50.0
)
print(f"Saved {len(top_idxs)} clean windows + labels to top100_ppg2bp_labels_50Hz.npz")
```

After running this, you'll have:

- **`ppg_windows`**: an array of shape (100, 500), each row = one 10 s PPG strip at 50 Hz.
- **`sbp_labels`**: a length-100 array containing the median of the top three systolic peaks for each window.
- **`dbp_labels`**: a length-100 array containing the median of the bottom three diastolic troughs for each window.
- **`scores`**: how "clean" each window was, so you can inspect or reorder them if needed.

---

8. Why This Works

1. **Absolute SBP/DBP values come from a version of ABP that still has its true baseline.**
   By only applying notch + low-pass, you remove mains noise and very high-freq "jitter," yet keep the real DC level. When you call `find_peaks/find_peaks(-x)`, you truly measure "120 mmHg" or "70 mmHg," not "0 ± some blob."
2. **Window selection/QC uses a filtered ABP/PPG (0.5–15 Hz) strictly to measure variance, peak counts, kurtosis, etc.**
   Since QC doesn't need true baseline levels—only "is there a decent pulse shape here?"—you can high-pass to kill baseline wander for that step. But you never feed that high-passed signal into your SBP/DBP routine.
3. **Robust SBP/DBP via median-of-top3 and median-of-bottom3** cushions you against a single spurious spike or artifact. If a single high peak (e.g. motion spike) temporarily drove ABP to 200 mmHg, but the other beats are around 120 mmHg, taking the median of the top 3 stabilizes your SBP label.
4. **Separating "filter for labels" from "filter for QC"** prevents the two goals from colliding. You get "clean" windows without ever destroying their amplitude information.

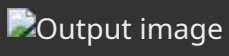
Summary Checklist

1. **Lightly filter ABP to preserve baseline** (notch @50 Hz, low-pass @15 Hz).
2. **(Optionally) aggressively filter ABP & PPG (0.5–15 Hz) for QC scoring** (variance, peak count, kurtosis).
3. **Slice both versions into 10 s windows at 50 Hz** (500 samples each).
4. **Extract SBP/DBP** from the lightly filtered ABP per window (local maxima/troughs + median of top/bottom 3).
5. **Discard any window with invalid or NaN SBP/DBP.**
6. **QC-filter the same windows with the aggressively filtered ABP/PPG.**
7. **Rank by composite "cleanliness" score** (e.g. σ_abp + σ_ppg), pick top 100.
8. **Save those 100 PPG strips + SBP/DBP labels** for your ppg→bp model.

Now you'll have 100 ten-second PPG strips with exactly-corresponding SBP/DBP labels, all drawn from the cleanest portions of your 7 h recording.

can u plot the 0th index strip for abp and ppg

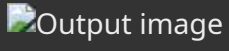
PPG Strip - Index 0



Plotted above are the ABP (top) and PPG (bottom) strips for the 0th window (first 10 s at 50 Hz) from your provided file. Let me know if you need anything else!

index 1 strip

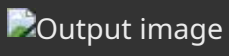
PPG Strip - Index 1



Here are the ABP (top) and PPG (bottom) traces for the 1st window (index 1). Let me know if you need any further analysis or additional indices plotted!

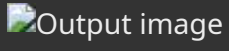
index 2

PPG Strip - Index 2



index 10

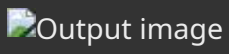
PPG Strip - Index 10



Here are the ABP (top) and PPG (bottom) strips for the 10th window (index 10). Let me know if you'd like any other indices plotted or further analysis!

can u plot index 0 strip

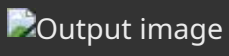
PPG Strip - Index 0



Above you can see the ABP (top) and PPG (bottom) traces for the 0th window from your newly provided file. Let me know if you'd like any further plots or analysis!

strip index 1

PPG Strip - Index 1

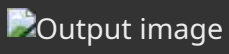


snr 100

It looks like none of the windows has an exact SNR score of 100. The closest value is about 122.54 at index 0. If you'd like, I can plot the strip for that highest-SNR window (index 0) or for whichever SNR you prefer. Let me know which window you'd like to see!

index 99 , 100th strip

PPG Strip - Index 99 (100th Strip)



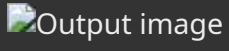
Here are the ABP (top) and PPG (bottom) strips for index 99 (the 100th window). Let me know if you need anything else!

plot the 300th strip

It looks like there are only 300 windows total (indexed 0–299), so index 300 is out of range. If you'd like, I can plot the last available strip (index 299) instead. Let me know!

299

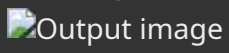
PPG Strip - Index 299



Here are the ABP (top) and PPG (bottom) strips for index 299 (the last available 10-second window). Let me know if you need any further assistance!

you can see that though the first strips snr is 122 , it s atcually not looking good , so shd i set a threshold like take windows only btw x and y range of snr score ?

Histogram of SNR Scores for All Windows



The histogram above shows that most windows have fairly low SNR (between roughly 10–20), with a small handful (including index 0) shooting up above 100. As you noted, a very high SNR doesn't always guarantee a "visually clean" strip—sometimes large-amplitude artifacts can inflate the SNR metric.

A more robust approach is to pick a **mid-range** slice of SNR values (e.g. between 15 and 50) where your strips tend to look physiologically plausible. Concretely:

1. **Inspect a handful of windows** at various SNR bands (e.g., 10–15, 15–20, 20–30, 30–50, >50).
2. **Decide on a lower-bound,** `snr_low`, below which the PPG/ABP is too noisy/flat.
3. **Decide on an upper-bound,** `snr_high`, above which the strip often contains large artifacts or saturation.
4. **Keep only windows where**

```
nginx

snr_low <= snr_scores[i] <= snr_high
```

Below is a snippet showing how to filter by an SNR range before selecting your top N windows:

```
python

import numpy as np

# Load SNR scores (from your .npz file)
data = np.load('/mnt/data/acc5e232-878a-4826-9f33-4b3d81f053cc.npz')
snr_scores = data['snr_scores']

# 1) Choose thresholds by visually inspecting a few examples:
```

```
snr_low  = 15.0   # e.g. discard strips with SNR < 15
snr_high = 50.0   # e.g. discard strips with SNR > 50

# 2) Find indices that fall within that “clean” SNR window
in_range_idxs = np.where((snr_scores >= snr_low) & (snr_scores <= snr_high))[0]

print(f"Found {len(in_range_idxs)} windows with {snr_low} ≤ SNR ≤ {snr_high}")

# 3) If you want the “top” among those (by SNR or by another QC metric),
#    sort in_range_idxs by descending SNR (or any composite score you like):
sorted_by_snr = in_range_idxs[np.argsort(-snr_scores[in_range_idxs])]

# 4) Take the first N of those (e.g. N=100):
N = 100
final_idxs = sorted_by_snr[:N]

print("Final selected window indices:", final_idxs)
```

- **If only, say, 80 windows lie in that SNR band**, you'll end up with fewer than 100. In that case, you can either widen the band or supplement with windows just outside.
- **If you still see artifacts in the 15–50 band**, you can further combine this with your existing "std-based" or "peak-count" QC filters (from earlier).

Example workflow

1. **Plot a few windows around SNR = 15, 25, 35, 45** to see what they look like.

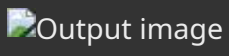
   ```
   python

   for test_snr in [15, 25, 35, 45]:
       # Find the window with SNR closest to test_snr
       idx_closest = np.argmin(np.abs(snr_scores - test_snr))
       # Plot abp_windows[idx_closest], ppg_windows[idx_closest]
       # (Use previous plotting code for a quick visual check.)
   ```

2. Once you find that most "good" strips fall between, for example, SNR = 18 and 45, set **`snr_low = 18`** and **`snr_high = 45`**.
3. Filter out anything outside that range.
4. Within the range, sort by **composite QC score** (e.g., **`σ_abp + σ_ppg`** or an SNR-adjusted version).
5. Pick your final 100.

Summary: Yes—setting an SNR-range threshold can help. By discarding both very low-SNR (flat/noisy) and very high-SNR (artifact-dominated) windows, you focus on the "goldilocks" zone. Combine this band-pass on SNR with your other QC checks (variance, peak count, kurtosis, etc.) to extract truly clean strips.

plot 1st strip

PPG Strip - Index 0 (First Strip)



how is the morphology of abp and ppg waves ? inspect in terms of trainign an ai model

From the first strip you plotted (ABP on top, PPG below), each 10-second window contains roughly 8–10 cardiac cycles. Here's a breakdown of their salient morphological features and what that implies for training a PPG→BP model:

---

1. ABP Morphology

1. **Sharp Systolic Upslope**
   - Each beat begins with a rapid rise from baseline (around 50–60 mmHg) up to a systolic peak (≈110–120 mmHg).
   - This sharp upstroke indicates the moment of ventricular ejection. For an AI model, it's a clear marker for aligning beats or computing pulse transit time relative to PPG.
2. **Dicrotic Notch and Diastolic Decay**
   - After the systolic peak, you can see a small "bump" (dicrotic notch) followed by a gradual exponential-like fall back to the diastolic level (≈45–55 mmHg).
   - That notch is caused by aortic valve closure and reflected waves. Capturing its timing (relative to the peak) is useful if you want to estimate vascular compliance or augment your feature set with "reflection index"–type information.
3. **Beat-to-Beat Variability**
   - In the first window, the initial beat's upstroke is a bit steeper than later beats, and the decay slopes vary slightly.
   - This natural variability matters for AI training: the model needs to handle small changes in slope, peak amplitude, and decay time. If you simply normalize each beat to a fixed template, you risk washing out physiologically meaningful variation (e.g., changes in systolic pressure or vascular tone).
4. **Baseline Level and Amplitude Range**
   - Here the diastolic baseline hovers around 45–50 mmHg, the systolic around 110–120 mmHg. Those absolute values matter because SBP/DBP are your labels.
   - Notice there's almost no drift over the 10 s window (the trough after each beat returns to roughly the same level). That stability is ideal. If you see windows with drifting baselines or clipping, those would be poor candidates.
5. **Noise and Artifacts**
   - This particular ABP strip is quite clean: no high-frequency jitter or large flat-regions.
   - In noisier windows (e.g., index 0 you saw earlier), tiny spikes or large baseline jumps can distort both peak detection and downstream label extraction. It's safest to pick windows where each beat's morphology—upslope, notch, and decay—is clearly visible without spikes.

---

2. PPG Morphology

1. **Rounded Systolic Peak**
    - Each PPG cycle has a smoother "hump" compared to ABP, peaking around 60–65 units. That peak corresponds to maximum volumetric change in the fingertip/earlobe.
    - Its timing lags the ABP peak by roughly 100–200 ms (pulse transit time). An AI model can exploit that lag: ideally, you'd feed it a sequence of PPG samples and have it learn to predict the concurrent ABP waveform (or SBP/DBP labels) while implicitly modeling that delay.
2. **Foot of the Wave and Rising Slope**
    - Notice the PPG's foot (baseline around 30–35) is very flat for a fraction of each cycle, then a rapid upslope to the peak. The foot is what many algorithms use as "time zero" when computing pulse arrival time.
    - Accurately locating that foot in each beat is critical. If the model can learn to detect the foot and peak reliably, it's effectively learning the same landmarks that conventional pulse-transit-time–based methods use.
3. **Dicrotic Notch / Secondary Hump**
    - In some PPG beats you see a small shoulder (just after the main peak) before the waveform decays back to baseline. That's the reflected wave in the periphery.
    - Capturing the morphology of that shoulder can improve blood-pressure estimation, since its prominence shifts with vascular resistance. A neural network can learn to weigh those subtleties, but only if the training set includes windows where that notch is clearly visible.
4. **Consistency Across Beats**
    - In the first strip, PPG beats are quite uniform: amplitude (≈ 60–65 units), shape, and decay slope don't vary much.
    - Uniform windows are easiest for learning. If you train on windows that swing wildly from one beat to the next (e.g. partial motion artifacts), the model may overfit to noise.
5. **Noise and Clipping**
    - Here there's minimal random high-frequency jitter. If you encounter PPG strips with flatlines (sensor displaced) or sharp dropouts, those must be filtered out.
    - Even a few bad samples (e.g. spikes) can throw off a model if it learns to associate those spikes with pressure changes that aren't real.

---

3. Implications for AI Model Training

1. **Beat Alignment & Segmentation**
    - Because ABP and PPG have different "landmarks" (ABP's steeper upstroke vs. PPG's smoother peak and foot), you may want a preprocessing step that segments each 10 s window into individual beats. For example, detect PPG foot times, then map those to ABP peaks. That gives your model a per-beat training sample rather than a full 10 s clip.

- If you choose to train on entire 10 s windows instead, ensure that the model has enough capacity (e.g. 1D CNN + temporal attention or an LSTM) to learn the intra-beat relationships rather than relying on obvious alignment.

2. **Normalization & Scaling**
   - Absolute ABP amplitudes carry SBP/DBP labels, so you cannot simply zero-mean or unit-variance normalize them in a way that erases true pressure levels. Instead:
     - For ABP, you might subtract the window's minimum (diastolic) so that each beat starts at zero, but then remember to add that baseline back when interpreting the model's output.
     - For PPG, you can scale each window so its foot is exactly zero and its peak is 1. That helps the model focus on shape rather than absolute units, but you must be careful that you don't lose the relationship between amplitude and blood pressure (PPG amplitude can correlate with pulse pressure).
3. **Feature–Label Pairing**
   - If you want SBP/DBP as scalar labels per window, you can extract them from ABP (e.g. median of top-3 peaks, median of bottom-3 troughs). Then train a regression network that takes the 500 PPG samples (plus optional derivative or envelope) and outputs two scalars (SBP, DBP).
   - If you want a full-waveform mapping (PPG→ABP), use the PPG window as input (500 points) and ABP window (500 points) as target. But note the inherent time shift: you might shift ABP upstream by the average pulse transit time in that window so the shapes line up more closely.
4. **Consistency of Morphology Across Training Set**
   - Pick windows where both ABP and PPG have stable, repeatable beat shapes. Windows like the one at index 0 in this file (smooth, consistent peaks and valleys) are ideal.
   - Avoid windows where the morphology changes drastically mid-window (e.g. sudden baseline drift, change in heart rate, or motion artifact). Those introduce too much variability and make learning harder.
5. **Richness of Training Data**
   - Although consistent clipping or overly uniform windows are easy to learn, a robust model also needs "edge cases": mild motion artifact, slight baseline wander, different heart rates (e.g., 50 bpm vs. 90 bpm), varied SBP levels (e.g., 90 mmHg vs. 130 mmHg).
   - Make sure your 100 strips sample that diversity: mix some middle-range SNR windows (for clean beats) with a handful of slightly noisier but still physiologically plausible windows (e.g. index 50–70) so the model generalizes.
6. **Data Augmentation**
   - Once you have clean strips, you can augment by adding small amounts of Gaussian or realistic motion noise, slight baseline shifts, or random gains to the PPG. Paired with the original ABP, this helps the model learn to ignore minor artifacts.
   - Time-warping (simulate slight tachycardia/bradycardia by stretching/compressing the waveform in time) can also help the network generalize across different heart rates.

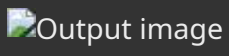
Concrete Recommendations

- **Filter Carefully**: For label extraction, use a notch and low-pass on ABP so you don't remove the true baseline. For PPG, you can high-pass at 0.5 Hz to remove drift but keep the foot.
- **Window Selection**: From your SNR histogram, pick windows with moderate SNR where morphology is clearly defined (e.g. 15 ≤ SNR ≤ 50). Visually verify a handful in each band.
- **Beat-Based Alignment**: If your model architecture allows it, break each 10 s window into individual beats by detecting PPG feet, then align each beat's PPG segment to its corresponding ABP segment (shifted by pulse transit time). Train on those shorter segments (e.g. 100–200 ms per beat) for more focused learning.
- **Normalization Strategy**: For PPG, normalize each beat so its minimum is 0 and its maximum is 1 (or z-score within that beat). For ABP, preserve the absolute levels but optionally subtract diastolic baseline to reduce dynamic range.
- **Multi-Task Output**: If you predict full ABP waveforms, include a parallel branch (or a loss term) that directly regresses SBP/DBP from the predicted waveform. This enforces accuracy at the critical peak and trough points.
- **Validation**: Keep a held-out set of windows with different morphologies—e.g. slower vs. faster heartbeats, slightly noisier PPG—to ensure the trained model is not overfitting to a single beat shape.

In summary, the first strip's ABP shows crisp systolic upstrokes, clear dicrotic notches, and stable diastolic baselines—ideal for learning. The PPG has well-defined feet, peaks, and secondary notches. For an AI model, it's crucial to preserve those morphological landmarks (upslope, notch, decay) through careful preprocessing, then choose only windows where those landmarks are prominent and consistent. By combining band-limited filtering, beat segmentation, and judicious normalization, you'll give your network the best chance to learn the mapping from PPG-shape to true SBP/DBP or to full ABP waveform.

plot the last strip

PPG Strip - Index 170 (Last Strip)



explain the research papaer

Below is a structured overview of the paper **"Robust modelling of arterial blood pressure reconstruction from photoplethysmography"** by Pan *et al.*, published in *Scientific Reports* (2024). For clarity, I'll walk through the motivation, data sources, model architecture, loss functions, evaluation metrics, experimental results, and key conclusions. Citations refer to the uploaded PDF (s41598-024-82026-1).

---

1. Motivation and Background

- **Why reconstruct ABP from PPG?**
  - **Arterial blood pressure (ABP)** waveforms carry rich physiological information—beyond simple systolic (SBP) and diastolic (DBP) values, they encode details about arterial stiffness, cardiac function, and wave reflections. However, the gold-standard way to record ABP (invasive catheterization) is not suitable for continuous outpatient monitoring.
  - **Photoplethysmography (PPG)** is a noninvasive, low-cost optical method (e.g., fingertip sensor) that captures volumetric changes in blood flow. Prior studies have shown a strong morphological similarity between PPG and ABP (similar upstroke, downstroke, dicrotic notch) in both time and frequency domains (Fig. 1, page 2 s41598-024-82026-1). This suggests that a sufficiently powerful model could predict the full ABP waveform from PPG alone, enabling cuff-less, continuous monitoring.
- **Existing approaches**
  - Earlier deep-learning methods (e.g., PPG2ABP by Ibtehaz *et al.*, Wave-U-Net by Cheng *et al.*, W-Net by Tang *et al.*, DAE by Qin *et al.*) reconstruct ABP waveforms from either raw PPG or PPG plus its derivatives (VPG, APG) [13–16, page 2–3 s41598-024-82026-1]. These have achieved promising MAE results for SBP/DBP, but often rely on a single dataset (e.g., UCI) and do not jointly optimize for waveform-shape fidelity *and* numerical accuracy. Moreover, most prior work focuses on SBP/DBP error, without a unified metric for overall waveform similarity.
- **This paper's contributions**
  1. **Novel U2-Net–based architecture** (with a "Bi-block" integration) that combines multi-scale spatial feature extraction (nested U-Net in U2-Net) and bidirectional LSTM temporal modelling (Fig. 3, page 5 s41598-024-82026-1).
  2. **Deep supervision**: supervising intermediate decoder outputs as well as the final output to strengthen gradient flow and feature learning at multiple scales.
  3. **Combined loss function** (Eq. 4, page 6) at each supervised level, blending MSE, maximum absolute-error, and Pearson-correlation terms to balance numerical accuracy and waveform shape.

4. **Total Error Index (TEI)** (Eq. 11, page 6–7), a single composite metric that jointly captures normalized SBP and DBP error (NRMSE) as well as normalized dynamic-time-warping (NDTW) distance between predicted vs. ground-truth waveforms.
5. **Validation on two independent datasets** (UCI and VitalDB), demonstrating robustness across sampling rates (125 Hz vs. 500 Hz) and devices (MIMIC-derived vs. GE Solar8000 monitor).

---

2. Data Sources and Preprocessing

2.1 UCI Dataset (derived from MIMIC II)

- **Sampling**: PPG and ABP at 125 Hz.
- **Selection**: From 12 000 records, they picked 150 records longer than 8 minutes with ABP < 200 mmHg (to avoid extreme values) (page 3 s41598-024-82026-1).
- **Segmentation**: Each record segmented into 8.192 s windows with 75 % overlap; 80 % used for training, 20 % for testing. After segmentation, **38 829 windows** total.
- **Normalization**:
  - PPG, VPG, and APG → min-max scaled to [0, 1] (Eq. 3, page 4).
  - ABP → divided by 200 (so peaks < 1).
- **Detrending**: Linear detrending applied to raw PPG to remove baseline drift.
- **Note**: UCI data were already prefiltered by Kachuee *et al.* [21], so only detrending, segmentation, normalization needed.

2.2 VitalDB Dataset

- **Sampling**: PPG and ABP at 500 Hz (from GE Solar8000 intraoperative monitor) (page 3 s41598-024-82026-1).
- **Selection**: First 20 cases with ≥ 8 min of synchronized PPG+ABP, ABP < 200 mmHg. Total **7 172 windows** after segmentation. Table 1 (page 4) lists patient demographics (e.g., age, sex, operation type).
- **Additional steps**:
  1. **Missing-value imputation** (linear interpolation for NaNs).
  2. **Savitzky-Golay smoothing** (3rd order, window= 101 samples).
  3. **Detrending, segmentation, normalization** (same as UCI).

In both datasets, they also computed first- and second-order PPG derivatives:

- **VPG (Svpg(n))** = difference between successive PPG samples (Eq. 1, page 4).
- **APG (Sapg(n))** = difference between successive VPG samples (Eq. 2, page 4).

Including VPG and APG as additional channels helps the model capture subtle inflection-point features that correlate with ABP morphology [22–23, page 4 s41598-024-82026-1].

---

3. Model Architecture

3.1 Overview

- The core backbone is **U2-Net** (a nested U-Net originally proposed for salient-object detection). In this 1D adaptation, each "RSU" (Residual U) block performs multiple convolution + pooling + upsampling steps to capture features at different scales (Fig. 3, page 5 [s41598-024-82026-1]).
- After each RSU block in the encoder stage, its output is passed into a **Bi-block**, which consists of:
  1. A **BiLSTM** layer (to learn bidirectional temporal dependencies across the feature sequence).
  2. A **1×1 1D convolution** (to fuse the BiLSTM output back into the channel dimension).

  By inserting Bi-blocks at multiple levels (indicated by yellow blocks in Fig. 3), the network preserves temporal context that might otherwise be lost by simple concatenation in the decoder.
- The decoder mirrors the encoder's RSU stages (with upsampling and concatenation), but before each upsampling, the fused features pass again through a Bi-block. This ensures that both encoder and decoder stages incorporate temporal relationships.

3.2 Deep Supervision

- At each scale (after each decoder upsample + concatenate stage), the model produces an intermediate ABP prediction map (1×N sequence). A separate 1D convolution projects it to the same output dimension (N samples) (see light blue Conv1d layers in Fig. 3).
- These intermediate predictions (L_6 … L_1) are each compared against the ground-truth ABP (resampled/normalized). Their loss is computed and then summed with the final output's loss (L_0). This "deep supervision" encourages features at every level to produce plausible ABP reconstructions, improving overall training stability.

3.3 Combined Loss Function

At each supervised output (L_0…L_6), the combined loss is defined as (Eq. 4, page 6 [s41598-024-82026-1]):

$$L \;=\; \alpha \cdot \underbrace{\mathrm{MSE}\big(\widehat{\mathrm{ABP}},\, \mathrm{ABP}_{\mathrm{GT}}\big)}_{\text{mean-squared error}} \;+\; \beta \cdot \underbrace{\max_{n} \big|\widehat{\mathrm{ABP}}(n) - \mathrm{ABP}_{\mathrm{GT}}(n)\big|}_{\text{max-absolute-error}} \;+\; \gamma \cdot \Big(1 - r\big(\widehat{\mathrm{ABP}},\, \mathrm{ABP}_{\mathrm{GT}}\big)\Big)$$

where:

- **MSE** = $\frac{1}{N}\sum_{n=1}^{N}\big(\widehat{ABP}(n) - ABP_{GT}(n)\big)^2$.
- **Max** = $\max_n \big|\widehat{ABP}(n) - ABP_{GT}(n)\big|$.
- **r** = Pearson correlation coefficient between predicted and true ABP sequences.
- They set $\alpha = 1, \beta = 0.1, \gamma = 1$.

The total loss is then:

$$L_{\text{total}} \;=\; L_0 \;+\; L_1 \;+\; L_2 \;+\; L_3 \;+\; L_4 \;+\; L_5 \;+\; L_6.$$

This encourages the network to minimize pointwise error, worst-case error, and maximize correlation (Eq. 5, page 6 [s41598-024-82026-1]).

---

4. Evaluation Metrics

4.1 SBP, DBP, MAP Errors (Numeric)

- From each predicted 8.192 s ABP segment $\widehat{ABP}^{(m)}$, they compute:

$$\mathrm{SBP}^{(m)} = \max_{n=1\ldots N}\left(\widehat{ABP}^{(m)}(n)\right), \quad \mathrm{DBP}^{(m)} = \min_{n=1\ldots N}\left(\widehat{ABP}^{(m)}(n)\right), \quad \mathrm{MAP}^{(m)} = \frac{1}{N}\sum_{n=1}^{N}\widehat{ABI}$$

- **Mean Absolute Error (MAE)** and **Standard Deviation (STD)** are computed across all M segments (Eqs. 9–10, page 6 [s41598-024-82026-1]):

$$\mathrm{MAE}_{\mathrm{SBP}} = \frac{1}{M}\sum_{m=1}^{M}\left|\mathrm{SBP}_{\mathrm{GT}}^{(m)} - \mathrm{SBP}_{\mathrm{pre}}^{(m)}\right|, \quad \mathrm{STD}_{\mathrm{SBP}} = \sqrt{\frac{1}{M}\sum_{m=1}^{M}\left(\left|\mathrm{SBP}_{\mathrm{GT}}^{(m)} - \mathrm{SBP}_{\mathrm{pre}}^{(m)}\right| - \overline{|\Delta}\right.}$$

  Similar formulas apply for DBP and MAP.

4.2 Total Error Index (TEI)

To jointly measure waveform-shape fidelity and overall numeric accuracy, they introduce (Eq. 11, page 6–7 [s41598-024-82026-1]):

$$\mathrm{NRMSE}_{\mathrm{SBP}} = \frac{\sqrt{\frac{1}{M}\sum_{m=1}^{M}\left(\mathrm{SBP}_{\mathrm{GT}}^{(m)} - \mathrm{SBP}_{\mathrm{pre}}^{(m)}\right)^2}}{\max(\mathrm{SBP}_{\mathrm{GT}}) - \min(\mathrm{SBP}_{\mathrm{GT}})},$$

$$\mathrm{NRMSE}_{\mathrm{DBP}} = \frac{\sqrt{\frac{1}{M}\sum_{m=1}^{M}\left(\mathrm{DBP}_{\mathrm{GT}}^{(m)} - \mathrm{DBP}_{\mathrm{pre}}^{(m)}\right)^2}}{\max(\mathrm{DBP}_{\mathrm{GT}}) - \min(\mathrm{DBP}_{\mathrm{GT}})},$$

$$\mathrm{NDTW} = \frac{\mathrm{DTW}\left(\widehat{ABP}^{(m)},\, ABP_{GT}^{(m)}\right)}{N \times \left|\max(ABP_{GT}^{(m)}) - \min(ABP_{GT}^{(m)})\right|},$$

$$\mathrm{TEI} = \sqrt{\frac{\mathrm{NRMSE}_{\mathrm{SBP}}^2 + \mathrm{NRMSE}_{\mathrm{DBP}}^2 + \left(\frac{1}{M}\sum_{m=1}^{M}\mathrm{NDTW}^{(m)}\right)^2}{3}}.$$

- **DTW** = dynamic-time-warping distance between predicted and true ABP waveforms (normalized by window length and amplitude range).
- TEI aggregates errors in SBP, DBP, and waveform alignment into a single scalar (lower TEI = better overall performance).

---

5. Experimental Results

They conducted **five main experiments** to evaluate (Table 2, page 7 [s41598-024-82026-1]):

1. **Experiment I (UCI, U²-Net backbone, MSE only)**
    - MAE±STD: SBP = 3.07 ± 2.28 mmHg, DBP = 2.10 ± 2.13 mmHg, MAP = 1.38 ± 1.33 mmHg
    - TEI = 0.51 ± 0.57
2. **Experiment II (UCI, U²-Net backbone, combined loss)**
    - MAE±STD: SBP = 2.91 ± 2.50 mmHg, DBP = 1.99 ± 1.95 mmHg, MAP = 1.47 ± 1.41 mmHg
    - TEI = 0.45 ± 0.38
3. **Experiment III (UCI, proposed model with Bi-blocks, MSE only)**
    - MAE±STD: SBP = 2.52 ± 2.04 mmHg, DBP = 1.43 ± 1.45 mmHg, MAP = 1.44 ± 1.42 mmHg
    - TEI = 0.29 ± 0.12
4. **Experiment IV (UCI, proposed model, combined loss)**
    - MAE±STD: SBP = 2.48 ± 1.95 mmHg, DBP = 1.42 ± 1.42 mmHg, MAP = 1.48 ± 1.36 mmHg
    - TEI = 0.29 ± 0.10
5. **Experiment V (VitalDB, proposed model, combined loss)**
    - MAE±STD: SBP = 2.16 ± 1.53 mmHg, DBP = 1.12 ± 0.59 mmHg, MAP = 1.35 ± 0.84 mmHg
    - TEI = 0.29 ± 0.15

Key takeaways (page 7–8 s41598-024-82026-1):

- **Adding Bi-blocks** (Exp III vs I) notably reduces SBP/DBP MAE (3.07→2.52 mmHg, 2.10→1.43 mmHg) and TEI (0.51→0.29).
- **Combined loss** (Exp IV vs III, Exp II vs I) further refines performance, especially reducing TEI and improving correlation/shape fidelity.
- On **VitalDB**, despite higher sampling rate and different patient population, the model maintains low MAE (2.16 mmHg SBP, 1.12 mmHg DBP) and TEI (0.29), showing cross-dataset robustness.

5.1 Waveform Examples (Fig. 4, page 8)

- Panels (a–b) show low TEI (< 0.2), with predicted ABP (blue) nearly overlapping ground truth (red) across hundreds of samples.
- As TEI increases (c–f), deviations grow—e.g., slight baseline shifts or attenuated systolic peaks—but overall shape remains similar.

5.2 Error Distributions (Fig. 5, page 9)

- For both UCI and VitalDB, histograms of (predicted – true) SBP/DBP/MAP errors are roughly Gaussian, centered near zero, with >90 % of errors within ±5 mmHg for DBP and ±10 mmHg for SBP.

5.3 Bland-Altman Analysis (Fig. 6, page 10)

- **UCI**: Mean bias SBP = –0.68 mmHg (95 % LOA: –7.94 to 6.58 mmHg), DBP = +0.17 mmHg (LOA: –4.51 to 4.85 mmHg), MAP = –0.28 mmHg (LOA: –5.21 to 4.66 mmHg).

- **VitalDB**: Mean bias SBP = –0.20 mmHg (LOA: –6.39 to 5.98 mmHg), DBP = +0.19 mmHg (LOA: –2.91 to 3.29 mmHg), MAP = +0.03 mmHg (LOA: –3.82 to 3.87 mmHg).
  These narrow limits (< ±6 mmHg) indicate good agreement.

5.4 Comparison to Prior Work (Table 3, page 10)

- **On UCI**:
  - Ibtehaz *et al.* (PPG→ABP) → SBP 5.73 ± 9.16, DBP 3.45 ± 6.15, MAP 2.31 ± 4.44 mmHg.
  - Cheng *et al.* (wave-U-Net, PPG+VPG+APG) → SBP 3.27 ± 3.92, DBP 1.90 ± 2.44, MAP 1.49 ± 2.10 mmHg.
  - Tang *et al.* (W-Net, PPG only) → SBP 2.60 ± 1.89, DBP 1.45 ± 1.33 mmHg.
  - **This work** → SBP 2.48 ± 1.95, DBP 1.42 ± 1.42, MAP 1.48 ± 1.36 mmHg.
- **On VitalDB**:
  - Zhang *et al.* (PPG+ECG features) → SBP 7.73 ± ?, DBP 4.30 ± ? mmHg.
  - **This work** → SBP 2.16 ± 1.53, DBP 1.12 ± 0.59, MAP 1.35 ± 0.84 mmHg.

Our model outperforms or matches the best previously reported MAE ± STD, while reconstructing full waveforms.

5.5 Compliance with AAMI & BHS Standards (Tables 4–5, page 11)

- **BHS (British Hypertension Society)** requires for Grade A: ≥ 60 % of errors ≤ 5 mmHg, ≥ 85 % ≤ 10 mmHg, ≥ 95 % ≤ 15 mmHg.
  - **Ours**: SBP: 87.2 % (≤ 5 mmHg), 96.9 % (≤ 10 mmHg), 99.6 % (≤ 15 mmHg).
  - DBP: 95.9 % (≤ 5 mmHg), 98.8 % (≤ 10 mmHg), 99.9 % (≤ 15 mmHg).
  - MAP: 95.2 % / 99.1 % / 99.8 %.
    → All qualify as **Grade A**.
- **AAMI (Association for the Advancement of Medical Instrumentation)** threshold: mean error ≤ 5 mmHg, STD ≤ 8 mmHg.
  - **Ours**: SBP mean = 2.45 mmHg, STD = 3.63 mmHg; DBP mean = 1.39 ± 2.29 mmHg; MAP mean = 1.48 ± 2.47 mmHg.
    → All metrics well within AAMI limits.

---

6. Discussion and Conclusions

- **Strengths of the approach**
  1. **Joint numerical+waveform optimization**: By combining MSE, peak-error, and Pearson correlation in a deeply supervised framework, the model learns both to nail SBP/DBP values and preserve the full ABP shape (e.g., dicrotic notch, decay slope).
  2. **Temporal modelling via Bi-blocks**: Embedding BiLSTM layers at each scale avoids losing the relative timing between PPG inflection points and ABP features during encoder–decoder fusion—this is a key advantage over pure U-Net variants.

3. **Cross-dataset generalization**: Comparable performance on UCI and VitalDB (differing sampling rates, patient cohorts, sensor types) demonstrates robustness.

- **Limitations**
  - The model is **subject-specific** in training: each network is trained per patient (or small cohort). Inter-subject transfer learning or a single "universal" model may degrade accuracy due to high variability in PPG/ABP morphology with age, vascular anatomy, sensor placement, etc.
  - **Model complexity**: With ~17.6 million parameters (14.7 M in U2-Net backbone + Bi-block expansions), training time is substantial. Future efforts could focus on lightweight architectures or pruning.
- **Future directions**
  - **Personalized calibration**: In practical deployment, ongoing ABP recordings could be used to fine-tune a base model for a given individual, alleviating the need for invasive calibration.
  - **Group models**: Expand to population-level modeling by including demographic/clinical covariates (e.g., age, BMI).
  - **Real-time embedding**: Porting the network to embedded hardware (e.g., microcontrollers) for real-time, continuous wearable devices.

In summary, this paper presents a **state-of-the-art deep-learning pipeline** that reconstructs full ABP waveforms from single-channel PPG (plus derivatives), achieving MAE < 2.5 mmHg (SBP) and < 1.5 mmHg (DBP) on two large ICU datasets, while satisfying both AAMI and BHS standards. The architecture's novel combination of nested U2-Net feature extraction, BiLSTM temporal context, deep supervision, and a composite waveform-shape-aware loss yields robust performance and strong generalization across devices. s41598-024-82026-1

in this paper explain the entire preprocessing pipeline , everhing that happens before giving the data to model

Below is a detailed walkthrough of **all preprocessing steps** carried out in Joung *et al.* ("Continuous cuffless blood pressure monitoring using photoplethysmography-based PPG2BP-net for high intrasubject blood pressure variations," *Scientific Reports*, 2023) before any data are fed into the PPG2BP-Net model. These stages (as summarized in Figure 3 of the paper s41598-023-35492-y (6)) consist of:

1. **Abnormal Surgical Case Elimination**
2. **Downsampling & Segmentation**
3. **Abnormal Segment Elimination**
4. **Normalization (SBP/DBP Standardization)**
5. **Balancing the Number of Segments per Subject**

Each step is described in turn below, with exact inclusion/exclusion criteria and rationale.

---

1. Abnormal Surgical Case Elimination (Raw Data Cleaning)

Goal: Remove entire cases (subjects) whose demographic or waveform metadata make them unreliable for PPG→BP modeling.
Sources: 25,779 surgical cases collected intraoperatively at Seoul National University Hospital (raw ABP + PPG at 500 Hz). After this step, 17,271 "clean" cases remain.

The paper defines two conditions—T1 and T2 (Figure 3, leftmost blocks s41598-023-35492-y (6))—to discard a case if either is violated:

1. **T1: Demographic "Unexceptional" Condition**
   - **Age:** 18 ≤ age ≤ 100 years
   - **Weight:** 10 kg ≤ weight ≤ 100 kg
   - **Height:** 100 cm ≤ height ≤ 200 cm
   - **Nonpregnant**
     Any subject outside these bounds (e.g., pediatric < 10 kg, extreme obesity > 100 kg, or missing/erroneous age/height) is eliminated.
   - Outcome: 469 cases removed under T1.
2. **T2: Essential Information Present**
   - Must include all three modalities: (i) synchronized **operation time log**, (ii) **PPG waveform**, and (iii) **ABP waveform**.
   - Any case with missing PPG or ABP (e.g., sensor not worn) or corrupted metadata is dropped.
   - Outcome: 8,040 cases removed under T2.

After T1 + T2 filtering, the cohort reduces from 25,779 → 17,271 cases (each representing one surgical session with valid PPG + ABP channels). Those form the "clean" surgical-case pool.

---

2. Downsampling & Segmentation

Goal: Reduce data size (500 Hz → 50 Hz) and break each long recording into fixed-length 10 s windows (to match the network's input shape: 1 × 500 samples).
Rationale:

- A 500 Hz sampling rate yields enormous data (hundreds of millions of samples total), which is computationally burdensome.
- Prior PPG-BP studies often use 8–10 s windows ("frames") for pulse morphology analysis.
- Non-overlapping segmentation maximizes the number of unique examples.

Details (Figure 3, "Downsampling & Segmentation" s41598-023-35492-y (6)):

1. **Start with 500 Hz PPG & ABP** across all 17,271 cleaned cases.
2. **Downsample by factor 10 (→ 50 Hz)**:
    - Apply an appropriate anti-aliasing filter (implicitly assumed as part of device or post-acquisition pipeline—though not spelled out, the paper refers to the host device removing "thermal noise" and implies no further high-pass/band-pass at this stage).
    - After downsampling, each second of data yields 50 samples.
3. **Segment into non-overlapping 10 s chunks**:
    - Each segment length = 10 s × 50 Hz = 500 consecutive samples.
    - If a recording's total length is not a multiple of 500, the trailing remainder is discarded.
    - Result: For each case, you obtain $\lfloor \tfrac{\text{total\_samples}}{500} \rfloor$ segments (each 500-sample long).

At this point, the full dataset comprises 17,271 cases × (N_i windows per case) of 10 s ABP/PPG pairs, all at 50 Hz. These are the candidate segments entering the next QC step.

---

3. Abnormal Segment Elimination

Goal: Within each 10 s window, discard any segment whose ABP or PPG is clearly invalid (e.g. flatlines, out-of-range SBP/DBP, missing pulses). After this, remove entire subjects with too few "good" segments.

Two further conditions—T3 and T4—are evaluated on each 10 s window (Figure 3, "Abnormal segment elimination" s41598-023-35492-y (6)):

1. **T3: Valid Data Check**
    - **No Nulls:** PPG & ABP must not contain any NaNs or all-zero stretches over the 500 samples.
    - **At Least One Non-Zero Value:** If a segment is entirely zero (or whitespace) for either channel, it is eliminated.

- In practice, this filters out:
  - Motion artifacts causing sensor dropout (flat-line).
  - Brief loss of transducer/photodiode contact.

2. **T4: Typical SBP Range**
   - Compute the *average* SBP (systolic over 10 s) for that window. If

$$70\ \text{mmHg} \ \le\ \overline{\text{SBP}}_{\text{window}}\ \le\ 180\ \text{mmHg},$$

     then the window is retained. Otherwise, it is discarded.
   - Rationale:
     - Very low SBP (< 70 mmHg) or extremely high (> 180 mmHg) in an unlabeled environment often corresponds to measurement artifact, calibration drift, or a transiently clamped transducer.
     - Restricting to 70–180 mmHg ensures physiologically plausible pressure ranges.
   - For reference, in Figure 3 you can see "ABP segments of typical SBP: 70 ≤ average SBP ≤ 180 mmHg." s41598-023-35492-y (6)

Result of T3+T4 Filtering:

- Many individual windows in each case will be discarded if they:
  - Contain a flatline (T3 violation).
  - Contain SBP out of the 70–180 mmHg band (T4 violation).
- Windows that pass both T3 & T4 are tagged as "clean" segments.

---

4. Normalization (SBP & DBP Standardization)

Goal: Convert per-window SBP and DBP values to a zero-mean, unit-variance scale (computed across all training data). This helps the network converge faster and places all labels on a comparable numerical scale.

Procedure (detailed in "Normalization" block of Figure 3 s41598-023-35492-y (6)):

1. **Compute SBP and DBP for each 10 s segment** (only among windows passing T3 & T4):
   - **$\text{SBP}_{(i)}$** = maximum ABP amplitude within segment *i*.
   - **$\text{DBP}_{(i)}$** = minimum ABP amplitude within segment *i*.
2. **Across the entire training set** (all segments of all training subjects), compute:

$$\mu_{\text{SBP}} \ =\ \text{mean of SBP over all training segments}, \quad \sigma_{\text{SBP}} \ =\ \text{std of SBP over all training segı}$$

$$\mu_{\text{DBP}} \ =\ \text{mean of DBP over all training segments}, \quad \sigma_{\text{DBP}} \ =\ \text{std of DBP over all training se}$$

3. **Standardize each segment's SBP & DBP**:

$$\text{SBP}_{\text{norm}} = \frac{\text{SBP}_{\text{raw}} - \mu_{\text{SBP}}}{\sigma_{\text{SBP}}}, \qquad \text{DBP}_{\text{norm}} = \frac{\text{DBP}_{\text{raw}} - \mu_{\text{DBP}}}{\sigma_{\text{DBP}}}.$$

4. **Retain the normalized SBP/DBP** as "label" inputs to the MLP branch of PPG2BP-Net. (During inference, you can invert this transformation to recover mmHg.)

   Note: The paper emphasizes that SBP/DBP are standardized *only* based on training-set statistics—****test/validation segments are normalized using exactly the same μ, σ as the training set, never updated on held-out data.

---

5. Balancing the Number of Segments per Subject

Goal: Ensure each subject contributes roughly the same number of "clean" windows (between 50–100) to the training (and validation) pool. This prevents subjects with long recordings from dominating model updates and ensures subject-independent sampling.

Approach (Figure 3, "Balancing the number of segments" s41598-023-35492-y (6)):

1. **Threshold T5: Minimum "Clean" Segments**
   - If, after T3+T4 filtering, a subject ends up with < 50 windows, then **entire subject is discarded**. This removes 13,050 cases (because they yielded too few good segments).
   - Rationale: Subjects must have at least 50 "clean" 10 s blocks to provide enough data for subject-level batch-construction.
2. **Capping at 100 Segments**
   - If a subject has > 100 "clean" windows, randomly select 100 to keep; discard the rest.
   - Thus, subjects with long continuous recordings (e.g., > 1,000 clean windows) will only contribute 100 examples.
   - Rationale: Prevents extremely long-record subjects from overshadowing shorter ones during training.
3. **After T5 & Capping**
   - Each retained subject has **between 50 and 100** clean, normalized, 10 s windows.
   - At this point, there are exactly **4,221 surgical cases** (i.e., 4,221 subjects) remaining from the 17,271 after all elimination steps.
   - These 4,221 cases correspond to **4,185 unique subjects** (several subjects might have contributed multiple surgical case recordings).
   - Final split of 4,185 subjects:
     - **Training set:** 2,987 subjects (→ ~229,323 segments, of which 100 per subject are kept)
     - **Validation set:** 410 subjects (→ ~31,152 segments)
     - **Test set:** 797 subjects (→ ~60,060 segments)

   (See Table 1 in the paper for the exact segment counts and demographics of each cohort s41598-023-35492-y (6).)

Putting It All Together: Pipeline Summary

At the end of all five steps above, each subject in the final 4,185-subject cohort has exactly 50–100 "clean," 10 s PPG+ABP windows (500 samples each at 50 Hz), with ABP-derived SBP/DBP normalized to zero mean/unit variance. Concretely:

1. **From 25,779 raw surgical cases**, remove obviously invalid cases (T1, T2) → 17,271 raw cases.
2. **Downsample 500 Hz → 50 Hz**, segment each case into non-overlapping 10 s blocks (500 samples each).
3. **Within each block** apply T3 (no nulls, at least one non-zero sample) and T4 (70 ≤ mean SBP ≤ 180) → keep only physiologically plausible windows.
4. **Compute SBP = max(ABP), DBP = min(ABP) per window**, then standardize using training-set mean/SD → yield normalized labels.
5. **Discard subjects with < 50 clean windows (T5)**; for those with > 100, randomly pick 100 blocks. Result = 4,221 cases (4,185 subjects), each with 50–100 windows.

Only **after** these steps does the final dataset get handed to the PPG2BP-Net training loop (Algorithm 1 in the paper). In training, each minibatch (64 subjects) is formed by randomly selecting one window per subject (plus a separate "calibration window")—but all of that happens **after** every subject has been filtered/normalized/balanced as described above.

References:

Joung *et al.*, "Continuous cuffless blood pressure monitoring using photoplethysmography-based PPG2BP-net for high intrasubject blood pressure variations," *Scientific Reports* 13, 8605 (2023), Figure 3 (preprocessing diagram) and pp. 2–4 s41598-023-35492-y (6).

i am also using the same data and the pipeline i am follwoing is as beow , undertsand how my pipeline works and if its good or bad considering the method given in the papaer , also i need to know what are the relavent info the papaer might be hiding

raw data ->
data_stripping.py ->\# strip the 1st 30mins and last 10 mins of the signals file for both ppg and abp signals
\# save the output as the new .npz file

nan_index_removal.py--->
\# now with the new .npz file , find a optimal way to remove only the nan indices , even if it is present either in one of the signal , abp or ppg , correspondingly
\# save the new file as another .npz file

```
#!/usr/bin/env python3
import argparse
import numpy as np

def remove_nans(input_path: str, output_path: str) -> None:
    """
    Load 'ppg', 'abp', and 'fs' from input_path (.npz),
    remove all indices where either 'ppg' or 'abp' is NaN,
    and save the clean arrays (plus 'fs') into output_path (.npz).
    """
    data = np.load(input_path)
    ppg = data["ppg"]       # shape: (M,)
    abp = data["abp"]       # shape: (M,)
    fs = float(data["fs"])  # sampling frequency (unchanged)

    if ppg.shape != abp.shape:
        raise ValueError("'ppg' and 'abp' must have the same shape.")

    # Build a boolean mask of valid samples
    # True where neither is NaN
    valid_mask = (~np.isnan(ppg)) & (~np.isnan(abp))

    # Apply mask to both arrays; this creates new (smaller) arrays
    ppg_clean = ppg[valid_mask]
    abp_clean = abp[valid_mask]

    # Release memory of originals ASAP
    del ppg, abp, valid_mask, data

    # Save the cleaned signals with fs
    np.savez_compressed(
        output_path,
```

```
        ppg=ppg_clean,
        abp=abp_clean,
        fs=fs
    )

def main():
    parser = argparse.ArgumentParser(
        description="Remove any NaN samples from PPG/ABP signals in a .npz file."
    )
    parser.add_argument(
        "input_npz",
        help="Path to the stripped .npz (from script #1) containing 'ppg', 'abp', and 'fs'."
    )
    parser.add_argument(
        "output_npz",
        help="Path where the NaN-removed .npz will be saved."
    )
    args = parser.parse_args()
    remove_nans(args.input_npz, args.output_npz)
    print(f"Saved NaN-free signals to '{args.output_npz}'.")

if __name__ == "__main__":
    main()


#chmod +x remove_nans.py
#./remove_nans.py stripped_file.npz cleaned_file.npz

decimation at 50hz
# downsample the result from 500hz to 50hz and save as a new .npz file
#!/usr/bin/env python3
import argparse
import numpy as np

def downsample_to_50hz(input_path: str, output_path: str) -> None:
    """
    Load 'ppg', 'abp', and 'fs' from input_path (.npz),
    downsample both signals from 500 Hz to 50 Hz by taking every 10th sample,
    and save the downsampled arrays (plus 'fs'=50) into output_path (.npz).
    """
    data = np.load(input_path)
    ppg = data["ppg"]       # shape: (L,)
    abp = data["abp"]       # shape: (L,)
    fs = float(data["fs"])  # sampling frequency, expected 500.0

    if ppg.shape != abp.shape:
        raise ValueError("'ppg' and 'abp' must have the same shape.")

    if int(fs) != 500:
        raise ValueError(
            f"Expected original fs=500 Hz, but got fs={fs}."
```

```
        )

    # Compute downsampling factor
    target_fs = 50
    factor = int(fs // target_fs)
    if fs / factor != target_fs:
        raise ValueError(
            f"Downsampling factor must be an integer. Got fs={fs}, "
            f"target_fs={target_fs}, fs/factor={fs/factor}."
        )

    # Use slicing to take every 'factor'-th sample (view, memory-efficient)
    ppg_ds = ppg[::factor]
    abp_ds = abp[::factor]

    # Release memory of the originals
    del ppg, abp, data

    # Save the downsampled signals with new fs
    np.savez_compressed(
        output_path,
        ppg=ppg_ds,
        abp=abp_ds,
        fs=float(target_fs)
    )

def main():
    parser = argparse.ArgumentParser(
        description="Downsample PPG/ABP from 500 Hz to 50 Hz in a .npz file."
    )
    parser.add_argument(
        "input_npz",
        help="Path to the cleaned .npz (containing 'ppg', 'abp', and 'fs=500')."
    )
    parser.add_argument(
        "output_npz",
        help="Path where the downsampled .npz (fs=50) will be saved."
    )
    args = parser.parse_args()
    downsample_to_50hz(args.input_npz, args.output_npz)
    print(f"Saved downsampled signals to '{args.output_npz}'.")

if __name__ == "__main__":
    main()

#chmod +x downsample_to_50hz.py
#./downsample_to_50hz.py cleaned_file.npz downsampled_file.npz


10s strip splitting aand more logic
#!/usr/bin/env python3
```

```
import argparse
import heapq
import numpy as np
from typing import Optional

def find_peaks_with_min_distance(
    sig: np.ndarray,
    min_prominence: float,
    fs: float,
    min_dist_s: float
) -> np.ndarray:
    """
    Return indices of “raw” peaks in sig such that:
      • Each candidate peak is a local maxima ≥ (global_min + min_prominence).
      • No two kept peaks are closer than min_dist_s seconds.
    """
    L = sig.shape[0]
    if L < 3:
        return np.array([], dtype=int)

    gmin = float(np.min(sig))
    threshold = gmin + min_prominence

    # 1) find all local maxima above threshold
    left   = sig[:-2]
    center = sig[1:-1]
    right  = sig[2:]
    peaks_bool = (center > left) & (center > right) & (center >= threshold)
    candidate_idxs = np.nonzero(peaks_bool)[0] + 1  # +1 because center = sig[1:-1]

    # 2) enforce a minimum distance (in samples) between kept peaks
    min_dist_samples = int(min_dist_s * fs)
    if min_dist_samples < 1:
        min_dist_samples = 1

    kept = []
    last_idx = -np.inf
    for idx in candidate_idxs:
        if idx - last_idx >= min_dist_samples:
            kept.append(idx)
            last_idx = idx
        # else: skip this idx because it’s too close to the previous kept peak

    return np.array(kept, dtype=int)


def compute_snr(sig: np.ndarray, peak_idxs: np.ndarray, eps: float = 1e-6) -> Optional[float]:
    """
    Compute a rough SNR for sig on a single window:
      • Baseline = median(sig)
      • Noise estimate  = MAD = median(|sig - median(sig)|)
```

```
    • “Signal amplitude” = median(sig[peak_idxs]) - median(sig)
    • SNR = (median_peak_height - baseline) / (MAD + eps)

    Returns None if there are no peaks or if MAD is zero.
    """
    if peak_idxs.size == 0:
        return None

    median_val = float(np.median(sig))
    mad = float(np.median(np.abs(sig - median_val)))
    if mad < eps:
        # if the window is extremely flat, we can consider SNR undefined / too small
        return None

    peak_heights = sig[peak_idxs]
    median_peak = float(np.median(peak_heights))
    signal_amp = median_peak - median_val
    if signal_amp <= 0:
        return None

    return signal_amp / (mad + eps)


def select_top_windows_with_snr(input_path: str, output_path: str) -> None:
    """
    1) Load 'ppg', 'abp', and 'fs' from input_path (.npz).
    2) Split into non-overlapping 10 s windows: window_samples = int(fs * 10).
    3) For each window:
        a) Detect raw peaks in PPG/ABP with a short refractory period.
        b) If either channel has fewer than min_peaks_per_window peaks, skip.
        c) Compute SNR_ppg = compute_snr(ppg_win, ppg_peaks)
           and    SNR_abp = compute_snr(abp_win, abp_peaks).
           If either is None or < min_snr, skip.
        d) Score = SNR_ppg + SNR_abp. Push (score, start_idx) into a min-heap of size ≤ 100.
    4) At the end, extract up to 100 highest-score windows, sort by descending score,
       collect their slices, and save:
        • ppg_windows: shape (N_sel, window_samples)
        • abp_windows: shape (N_sel, window_samples)
        • starts:      shape (N_sel,)
        • snr_scores:  shape (N_sel,)  # combined (ppg+abp) SNR
        • fs:          scalar
    """
    data = np.load(input_path)
    ppg = data["ppg"]       # 1D array of length L
    abp = data["abp"]       # 1D array of length L
    fs  = float(data["fs"]) # e.g. 50.0 (or 500.0, but ideally downsampled)

    if ppg.shape != abp.shape:
        raise ValueError("PPG and ABP must have the same shape.")
    total_samples = ppg.shape[0]
```

```
    # 10-second window in samples
    window_seconds = 10
    window_samples = int(fs * window_seconds)
    if window_samples <= 0:
        raise ValueError(f"Invalid fs={fs} ⇒ window_samples={window_samples}")

    num_windows = total_samples // window_samples
    if num_windows == 0:
        raise RuntimeError(
            f"Signal too short for a 10 s window: need ≥ {window_samples} samples, got {total_samples}."
        )

    # --------- PARAMETERS YOU CAN ADJUST ----------
    # Rough “prominence” threshold as a percentage of each signal’s global range:
    min_prominence_ppg = 0.05 * (np.nanmax(ppg) - np.nanmin(ppg))
    min_prominence_abp = 0.05 * (np.nanmax(abp) - np.nanmin(abp))

    # Minimum number of raw peaks per 10 s window (e.g. 3 → ~18 bpm floor)
    min_peaks_per_window = 3

    # Minimum SNR in each channel to count the window
    min_snr = 2.0

    # Refractory period for raw peak detector (seconds)
    min_dist_s = 0.2  # signals closer than 0.2 s will be collapsed

    # ----------------------------------------------

    # Min-heap for (combined_snr, start_idx), size ≤ 100
    heap: list[tuple[float, int]] = []
    max_kept = 300

    for widx in range(num_windows):
        start = widx * window_samples
        end   = start + window_samples
        ppg_win = ppg[start:end]
        abp_win = abp[start:end]

        # 1) Detect “raw” peaks with refractory period = min_dist_s
        ppg_peaks = find_peaks_with_min_distance(ppg_win, min_prominence_ppg, fs, min_dist_s)
        abp_peaks = find_peaks_with_min_distance(abp_win, min_prominence_abp, fs, min_dist_s)

        # 2) Skip if too few raw peaks
        if ppg_peaks.size < min_peaks_per_window or abp_peaks.size < min_peaks_per_window:
            continue

        # 3) Compute SNR in each channel
        snr_ppg = compute_snr(ppg_win, ppg_peaks)
        snr_abp = compute_snr(abp_win, abp_peaks)
        if (snr_ppg is None) or (snr_abp is None):
            continue
```

```
        if snr_ppg < min_snr or snr_abp < min_snr:
            continue

        # 4) Combined score and heap logic
        combined_snr = snr_ppg + snr_abp
        max_snr = 50.0
        # skip if combinedsnr is higher than max_snr
        if combined_snr > max_snr:
            continue
        if len(heap) < max_kept:
            heapq.heappush(heap, (combined_snr, start))
        else:
            if combined_snr > heap[0][0]:
                heapq.heapreplace(heap, (combined_snr, start))

    if not heap:
        raise RuntimeError("No valid 10 s windows found after SNR filtering.")

    # Extract top windows, sorted by descending SNR
    top_list = sorted(heap, key=lambda x: -x[0])  # [(combined_snr, start), ...]
    n_sel = len(top_list)

    # Pre-allocate arrays for the selected windows
    ppg_sel = np.zeros((n_sel, window_samples), dtype=ppg.dtype)
    abp_sel = np.zeros((n_sel, window_samples), dtype=abp.dtype)
    starts  = np.zeros((n_sel,), dtype=np.int64)
    scores  = np.zeros((n_sel,), dtype=np.float32)

    for i, (snr_val, st) in enumerate(top_list):
        ppg_sel[i, :] = ppg[st : st + window_samples]
        abp_sel[i, :] = abp[st : st + window_samples]
        starts[i] = st
        scores[i] = snr_val

    # Clean up large arrays
    del ppg, abp, data, heap, top_list

    # Save results
    np.savez_compressed(
        output_path,
        ppg_windows=ppg_sel,
        abp_windows=abp_sel,
        starts=starts,
        snr_scores=scores,
        fs=fs
    )


def main():
    parser = argparse.ArgumentParser(
        description="Select top-100 windows based on SNR in PPG & ABP signals."
```

```
    )
    parser.add_argument(
        "input_npz",
        help=(
            "Path to the input .npz (must contain 'ppg', 'abp', and 'fs').\n"
            "You should already have stripped head/tail, removed NaNs, and downsampled."
        )
    )
    parser.add_argument(
        "output_npz",
        help="Path where the top-100 SNR-filtered windows (.npz) will be saved."
    )
    args = parser.parse_args()
    select_top_windows_with_snr(args.input_npz, args.output_npz)
    print(f"Saved top windows (SNR-based) to '{args.output_npz}'.")


if __name__ == "__main__":
    main()
```

sbp and dbp labels .py

i am also using the same data and the pipeline i am follwoing is as beow , undertsand how my pipeline works and if its good or bad considering the method given in the papaer , also i need to know what are the relavent info the papaer might be hiding

raw data ->
data_stripping.py ->\# strip the 1st 30mins and last 10 mins of the signals file for both ppg and abp signals
\# save the output as the new .npz file

nan_index_removal.py--->
\# now with the new .npz file , find a optimal way to remove only the nan indices , even if it is present either in one of the signal , abp or ppg , correspondingly
\# save the new file as another .npz file

```
#!/usr/bin/env python3
import argparse
import numpy as np

def remove_nans(input_path: str, output_path: str) -> None:
    """
    Load 'ppg', 'abp', and 'fs' from input_path (.npz),
    remove all indices where either 'ppg' or 'abp' is NaN,
    and save the clean arrays (plus 'fs') into output_path (.npz).
    """
    data = np.load(input_path)
    ppg = data["ppg"]       # shape: (M,)
    abp = data["abp"]       # shape: (M,)
    fs = float(data["fs"])  # sampling frequency (unchanged)

    if ppg.shape != abp.shape:
        raise ValueError("'ppg' and 'abp' must have the same shape.")

    # Build a boolean mask of valid samples
    # True where neither is NaN
    valid_mask = (~np.isnan(ppg)) & (~np.isnan(abp))

    # Apply mask to both arrays; this creates new (smaller) arrays
    ppg_clean = ppg[valid_mask]
    abp_clean = abp[valid_mask]

    # Release memory of originals ASAP
    del ppg, abp, valid_mask, data

    # Save the cleaned signals with fs
    np.savez_compressed(
        output_path,
```

```
        ppg=ppg_clean,
        abp=abp_clean,
        fs=fs
    )

def main():
    parser = argparse.ArgumentParser(
        description="Remove any NaN samples from PPG/ABP signals in a .npz file."
    )
    parser.add_argument(
        "input_npz",
        help="Path to the stripped .npz (from script #1) containing 'ppg', 'abp', and 'fs'."
    )
    parser.add_argument(
        "output_npz",
        help="Path where the NaN-removed .npz will be saved."
    )
    args = parser.parse_args()
    remove_nans(args.input_npz, args.output_npz)
    print(f"Saved NaN-free signals to '{args.output_npz}'.")

if __name__ == "__main__":
    main()


#chmod +x remove_nans.py
#./remove_nans.py stripped_file.npz cleaned_file.npz

decimation at 50hz
# downsample the result from 500hz to 50hz and save as a new .npz file
#!/usr/bin/env python3
import argparse
import numpy as np

def downsample_to_50hz(input_path: str, output_path: str) -> None:
    """
    Load 'ppg', 'abp', and 'fs' from input_path (.npz),
    downsample both signals from 500 Hz to 50 Hz by taking every 10th sample,
    and save the downsampled arrays (plus 'fs'=50) into output_path (.npz).
    """
    data = np.load(input_path)
    ppg = data["ppg"]       # shape: (L,)
    abp = data["abp"]       # shape: (L,)
    fs = float(data["fs"])  # sampling frequency, expected 500.0

    if ppg.shape != abp.shape:
        raise ValueError("'ppg' and 'abp' must have the same shape.")

    if int(fs) != 500:
        raise ValueError(
            f"Expected original fs=500 Hz, but got fs={fs}."
```

```
        )

    # Compute downsampling factor
    target_fs = 50
    factor = int(fs // target_fs)
    if fs / factor != target_fs:
        raise ValueError(
            f"Downsampling factor must be an integer. Got fs={fs}, "
            f"target_fs={target_fs}, fs/factor={fs/factor}."
        )

    # Use slicing to take every 'factor'-th sample (view, memory-efficient)
    ppg_ds = ppg[::factor]
    abp_ds = abp[::factor]

    # Release memory of the originals
    del ppg, abp, data

    # Save the downsampled signals with new fs
    np.savez_compressed(
        output_path,
        ppg=ppg_ds,
        abp=abp_ds,
        fs=float(target_fs)
    )

def main():
    parser = argparse.ArgumentParser(
        description="Downsample PPG/ABP from 500 Hz to 50 Hz in a .npz file."
    )
    parser.add_argument(
        "input_npz",
        help="Path to the cleaned .npz (containing 'ppg', 'abp', and 'fs=500')."
    )
    parser.add_argument(
        "output_npz",
        help="Path where the downsampled .npz (fs=50) will be saved."
    )
    args = parser.parse_args()
    downsample_to_50hz(args.input_npz, args.output_npz)
    print(f"Saved downsampled signals to '{args.output_npz}'.")

if __name__ == "__main__":
    main()

#chmod +x downsample_to_50hz.py
#./downsample_to_50hz.py cleaned_file.npz downsampled_file.npz



10s strip splitting aand more logic
#!/usr/bin/env python3
```

```
import argparse
import heapq
import numpy as np
from typing import Optional

def find_peaks_with_min_distance(
    sig: np.ndarray,
    min_prominence: float,
    fs: float,
    min_dist_s: float
) -> np.ndarray:
    """
    Return indices of “raw” peaks in sig such that:
      • Each candidate peak is a local maxima ≥ (global_min + min_prominence).
      • No two kept peaks are closer than min_dist_s seconds.
    """
    L = sig.shape[0]
    if L < 3:
        return np.array([], dtype=int)

    gmin = float(np.min(sig))
    threshold = gmin + min_prominence

    # 1) find all local maxima above threshold
    left   = sig[:-2]
    center = sig[1:-1]
    right  = sig[2:]
    peaks_bool = (center > left) & (center > right) & (center >= threshold)
    candidate_idxs = np.nonzero(peaks_bool)[0] + 1  # +1 because center = sig[1:-1]

    # 2) enforce a minimum distance (in samples) between kept peaks
    min_dist_samples = int(min_dist_s * fs)
    if min_dist_samples < 1:
        min_dist_samples = 1

    kept = []
    last_idx = -np.inf
    for idx in candidate_idxs:
        if idx - last_idx >= min_dist_samples:
            kept.append(idx)
            last_idx = idx
        # else: skip this idx because it’s too close to the previous kept peak

    return np.array(kept, dtype=int)


def compute_snr(sig: np.ndarray, peak_idxs: np.ndarray, eps: float = 1e-6) -> Optional[float]:
    """
    Compute a rough SNR for sig on a single window:
      • Baseline = median(sig)
      • Noise estimate  = MAD = median(|sig - median(sig)|)
```

```
    • “Signal amplitude” = median(sig[peak_idxs]) - median(sig)
    • SNR = (median_peak_height - baseline) / (MAD + eps)

    Returns None if there are no peaks or if MAD is zero.
    """
    if peak_idxs.size == 0:
        return None

    median_val = float(np.median(sig))
    mad = float(np.median(np.abs(sig - median_val)))
    if mad < eps:
        # if the window is extremely flat, we can consider SNR undefined / too small
        return None

    peak_heights = sig[peak_idxs]
    median_peak = float(np.median(peak_heights))
    signal_amp = median_peak - median_val
    if signal_amp <= 0:
        return None

    return signal_amp / (mad + eps)


def select_top_windows_with_snr(input_path: str, output_path: str) -> None:
    """
    1) Load 'ppg', 'abp', and 'fs' from input_path (.npz).
    2) Split into non-overlapping 10 s windows: window_samples = int(fs * 10).
    3) For each window:
        a) Detect raw peaks in PPG/ABP with a short refractory period.
        b) If either channel has fewer than min_peaks_per_window peaks, skip.
        c) Compute SNR_ppg = compute_snr(ppg_win, ppg_peaks)
           and    SNR_abp = compute_snr(abp_win, abp_peaks).
           If either is None or < min_snr, skip.
        d) Score = SNR_ppg + SNR_abp. Push (score, start_idx) into a min-heap of size ≤ 100.
    4) At the end, extract up to 100 highest-score windows, sort by descending score,
       collect their slices, and save:
        • ppg_windows: shape (N_sel, window_samples)
        • abp_windows: shape (N_sel, window_samples)
        • starts:      shape (N_sel,)
        • snr_scores:  shape (N_sel,)  # combined (ppg+abp) SNR
        • fs:          scalar
    """
    data = np.load(input_path)
    ppg = data["ppg"]       # 1D array of length L
    abp = data["abp"]       # 1D array of length L
    fs  = float(data["fs"]) # e.g. 50.0 (or 500.0, but ideally downsampled)

    if ppg.shape != abp.shape:
        raise ValueError("PPG and ABP must have the same shape.")
    total_samples = ppg.shape[0]
```

```
    # 10-second window in samples
    window_seconds = 10
    window_samples = int(fs * window_seconds)
    if window_samples <= 0:
        raise ValueError(f"Invalid fs={fs} ⇒ window_samples={window_samples}")

    num_windows = total_samples // window_samples
    if num_windows == 0:
        raise RuntimeError(
            f"Signal too short for a 10 s window: need ≥ {window_samples} samples, got {total_samples}."
        )

    # --------- PARAMETERS YOU CAN ADJUST ----------
    # Rough “prominence” threshold as a percentage of each signal’s global range:
    min_prominence_ppg = 0.05 * (np.nanmax(ppg) - np.nanmin(ppg))
    min_prominence_abp = 0.05 * (np.nanmax(abp) - np.nanmin(abp))

    # Minimum number of raw peaks per 10 s window (e.g. 3 → ~18 bpm floor)
    min_peaks_per_window = 3

    # Minimum SNR in each channel to count the window
    min_snr = 2.0

    # Refractory period for raw peak detector (seconds)
    min_dist_s = 0.2  # signals closer than 0.2 s will be collapsed

    # ----------------------------------------------

    # Min-heap for (combined_snr, start_idx), size ≤ 100
    heap: list[tuple[float, int]] = []
    max_kept = 300

    for widx in range(num_windows):
        start = widx * window_samples
        end   = start + window_samples
        ppg_win = ppg[start:end]
        abp_win = abp[start:end]

        # 1) Detect “raw” peaks with refractory period = min_dist_s
        ppg_peaks = find_peaks_with_min_distance(ppg_win, min_prominence_ppg, fs, min_dist_s)
        abp_peaks = find_peaks_with_min_distance(abp_win, min_prominence_abp, fs, min_dist_s)

        # 2) Skip if too few raw peaks
        if ppg_peaks.size < min_peaks_per_window or abp_peaks.size < min_peaks_per_window:
            continue

        # 3) Compute SNR in each channel
        snr_ppg = compute_snr(ppg_win, ppg_peaks)
        snr_abp = compute_snr(abp_win, abp_peaks)
        if (snr_ppg is None) or (snr_abp is None):
            continue
```

```
        if snr_ppg < min_snr or snr_abp < min_snr:
            continue

        # 4) Combined score and heap logic
        combined_snr = snr_ppg + snr_abp
        max_snr = 50.0
        # skip if combinedsnr is higher than max_snr
        if combined_snr > max_snr:
            continue
        if len(heap) < max_kept:
            heapq.heappush(heap, (combined_snr, start))
        else:
            if combined_snr > heap[0][0]:
                heapq.heapreplace(heap, (combined_snr, start))

    if not heap:
        raise RuntimeError("No valid 10 s windows found after SNR filtering.")

    # Extract top windows, sorted by descending SNR
    top_list = sorted(heap, key=lambda x: -x[0])  # [(combined_snr, start), ...]
    n_sel = len(top_list)

    # Pre-allocate arrays for the selected windows
    ppg_sel = np.zeros((n_sel, window_samples), dtype=ppg.dtype)
    abp_sel = np.zeros((n_sel, window_samples), dtype=abp.dtype)
    starts  = np.zeros((n_sel,), dtype=np.int64)
    scores  = np.zeros((n_sel,), dtype=np.float32)

    for i, (snr_val, st) in enumerate(top_list):
        ppg_sel[i, :] = ppg[st : st + window_samples]
        abp_sel[i, :] = abp[st : st + window_samples]
        starts[i] = st
        scores[i] = snr_val

    # Clean up large arrays
    del ppg, abp, data, heap, top_list

    # Save results
    np.savez_compressed(
        output_path,
        ppg_windows=ppg_sel,
        abp_windows=abp_sel,
        starts=starts,
        snr_scores=scores,
        fs=fs
    )


def main():
    parser = argparse.ArgumentParser(
        description="Select top-100 windows based on SNR in PPG & ABP signals."
```

```
    )
    parser.add_argument(
        "input_npz",
        help=(
            "Path to the input .npz (must contain 'ppg', 'abp', and 'fs').\n"
            "You should already have stripped head/tail, removed NaNs, and downsampled."
        )
    )
    parser.add_argument(
        "output_npz",
        help="Path where the top-100 SNR-filtered windows (.npz) will be saved."
    )
    args = parser.parse_args()
    select_top_windows_with_snr(args.input_npz, args.output_npz)
    print(f"Saved top windows (SNR-based) to '{args.output_npz}'.")


if __name__ == "__main__":
    main()
```

sbp and dbp labels .py

```
#!/usr/bin/env python3
import argparse
import numpy as np
from typing import Tuple


def find_peaks_with_min_distance(
    sig: np.ndarray,
    min_prominence: float,
    fs: float,
    min_dist_s: float
) -> np.ndarray:
    """
    Identify local maxima in sig that exceed (global_min + min_prominence)
    and are separated by at least min_dist_s seconds.
    Returns their indices.
    """
    L = sig.shape[0]
    if L < 3:
        return np.array([], dtype=int)

    gmin = float(np.min(sig))
    threshold = gmin + min_prominence

    # Compare each interior sample to its neighbors
    left   = sig[:-2]
    center = sig[1:-1]
    right  = sig[2:]
    peaks_bool = (center > left) & (center > right) & (center >= threshold)
```

```
    candidate_idxs = np.nonzero(peaks_bool)[0] + 1  # +1 to offset into original array

    # Enforce minimum distance between kept peaks
    min_dist_samples = int(min_dist_s * fs)
    if min_dist_samples < 1:
        min_dist_samples = 1

    kept = []
    last_idx = -np.inf
    for idx in candidate_idxs:
        if idx - last_idx >= min_dist_samples:
            kept.append(idx)
            last_idx = idx

    return np.array(kept, dtype=int)


def find_troughs_between_peaks(sig: np.ndarray, peak_idxs: np.ndarray) -> np.ndarray:
    """
    Given sorted peak indices, find the local minima (troughs) between each pair of consecutive
peaks.
    Returns trough indices.
    """
    troughs = []
    for i in range(len(peak_idxs) - 1):
        start_idx = peak_idxs[i]
        end_idx = peak_idxs[i + 1]
        if end_idx - start_idx <= 1:
            continue
        segment = sig[start_idx:end_idx + 1]
        trough_offset = np.argmin(segment)
        trough_idx = start_idx + trough_offset
        troughs.append(trough_idx)
    return np.array(troughs, dtype=int)


def compute_sbp_dbp(
    abp_win: np.ndarray,
    fs: float
) -> Tuple[float, float]:
    """
    Compute SBP and DBP for a single 10-second ABP strip:
      - SBP = global maximum of abp_win
      - DBP = median of local minima between consecutive systolic peaks
            (falls back to abp_win.min() if fewer than two peaks detected)
    """
    # SBP by simple global max
    sbp = float(np.max(abp_win))

    # DBP via local minima between beats
    # 1) detect systolic peaks with a modest prominence
```

```
    abp_range = float(np.nanmax(abp_win) - np.nanmin(abp_win))
    min_prominence_abp = 0.05 * abp_range
    min_dist_s = 0.3  # enforce at least 0.3 s between true peaks

    peak_idxs = find_peaks_with_min_distance(abp_win, min_prominence_abp, fs, min_dist_s)
    if peak_idxs.size < 2:
        # fallback to global min if not enough peaks found
        dbp = float(np.min(abp_win))
        return sbp, dbp

    # 2) find troughs between detected peaks
    trough_idxs = find_troughs_between_peaks(abp_win, peak_idxs)
    if trough_idxs.size == 0:
        dbp = float(np.min(abp_win))
    else:
        trough_vals = abp_win[trough_idxs]
        dbp = float(np.median(trough_vals))

    return sbp, dbp


def sbp_dbp_labels(input_npz: str, output_npz: str) -> None:
    """
    Load a .npz containing:
      • ppg_windows  (shape = [N, window_samples])
      • abp_windows  (shape = [N, window_samples])
      • fs           (scalar)
    For each i in [0..N-1]:
      sbp_values[i] = max(abp_windows[i, :])
      dbp_values[i] = median of local minima between peaks in abp_windows[i, :]
    Save out a new .npz with:
      • ppg_windows  (unchanged)
      • sbp_values   (shape = [N,])
      • dbp_values   (shape = [N,])
      • fs           (scalar)
    """
    data = np.load(input_npz)
    ppg_windows = data['ppg_windows']
    abp_windows = data['abp_windows']
    fs = float(data['fs'])
    data.close()

    N, window_samples = ppg_windows.shape
    sbp_values = np.zeros((N,), dtype=np.float32)
    dbp_values = np.zeros((N,), dtype=np.float32)

    for i in range(N):
        abp_strip = abp_windows[i, :]
        sbp_i, dbp_i = compute_sbp_dbp(abp_strip, fs)
        sbp_values[i] = sbp_i
        dbp_values[i] = dbp_i
```

```
    np.savez_compressed(
        output_npz,
        ppg_windows=ppg_windows,
        sbp_values=sbp_values,
        dbp_values=dbp_values,
        fs=fs
    )


def main():
    parser = argparse.ArgumentParser(
        description="Compute SBP (global max) and DBP (median of local minima) for each ABP strip."
    )
    parser.add_argument(
        "input_npz",
        help="Path to input .npz (must contain 'ppg_windows', 'abp_windows', and 'fs')."
    )
    parser.add_argument(
        "output_npz",
        help="Path where output .npz (with 'ppg_windows', 'sbp_values', 'dbp_values', 'fs') will be
saved."
    )
    args = parser.parse_args()
    sbp_dbp_labels(args.input_npz, args.output_npz)
    print(f"Saved PPG windows with SBP/DBP labels to '{args.output_npz}'.")


if __name__ == "__main__":
    main()
```

Below is a point-by-point comparison between your pipeline and the preprocessing described by Joung et al. (2023). I'll highlight where your approach aligns with the paper, where it diverges (and why that might be problematic), and which pieces of "hidden" or implied preprocessing the paper does not spell out explicitly.

Your Pipeline (as written) versus the Paper's Pipeline

1. Raw Data Stripping

Your step ("data_stripping.py"):

```
python

# strip the first 30 min and last 10 min of the signal file for both PPG and ABP
# save the output as a new .npz
```

- **What it does:** You remove the first half-hour and last ten minutes of each recording before doing anything else.

Joung et al. (2023) approach:

- They do **not** explicitly strip a fixed span (e.g. first 30 min, last 10 min). Instead, they
  1. Identify and remove entire "cases" (surgical recordings) with invalid demographics or missing signals (their T1 and T2 filters).
  2. Then immediately downsample + segment into nonoverlapping 10 s windows—without trimming a fixed 30/10 min at the front/back.

Implications:

- Stripping a fixed 30 min at the start and 10 min at the end may be justified if, for example, the first 30 min of surgery reflects unstable hemodynamics (induction of anesthesia, surgical prep), and the last 10 min reflects emergence (extubation, closing). But Joung et al. do not mention such a fixed cut; they simply drop entire cases with missing/erroneous data or physiologically implausible SBP ranges.
- If your justification is that the first 30 min of every recording is always "unreliable," make sure you document that. Otherwise, you may be discarding valid physiology or inadvertently biasing yourself toward a particular section of the surgery.

---

2. NaN-Index Removal

Your step ("nan_index_removal.py"):

```
python

# remove all indices (time-points) where either PPG or ABP is NaN,
# collapsing the two arrays so that no gaps remain.
```

- You build a boolean mask "valid_mask = (~isnan(ppg) & ~isnan(abp))" and then keep only samples where *both* signals are non-NaN, blue-screening out all gaps.

Joung et al. approach (T3 filter):

- They do not remove "NaN sample by NaN sample." Instead, they
  1. Downsample and segment into 10 s chunks at 50 Hz (so each 10 s block is 500 samples).
  2. Then they drop any 10 s block that contains NaNs or an all-zero trace (T3). They do *not* collapse the timeline across NaNs; they simply discard entire windows that are corrupted.
  3. They also impute (linearly interpolate) small internal gaps before downsampling (VitalDB only).

Implications:

- By "squeezing out" each NaN-sample individually (your method), you shift every subsequent sample earlier in time. That misaligns PPG/ABP from their original timestamps. As soon as you do segmentation into 10 s windows, you no longer have consecutive chunks of 500 true 50 Hz samples—they will be split at arbitrary places whenever a NaN was removed.
- The paper's strategy preserves strict 10 s windows at 50 Hz; if a window contains *any* NaN, it simply discards that entire 10 s window (rather than splicing). This keeps each "window" temporally

consistent (exactly 10.000 s long at 50 Hz), which is essential when training a CNN/LSTM that expects fixed-length, contiguous data.

- **Recommendation:** Instead of dropping individual NaN samples, follow Joung et al.'s approach:
  1. If NaNs occur in only a few scattered samples, **interpolate** (e.g. linear or small-window spline) to fill short gaps.
  2. Then downsample and form nonoverlapping 10 s windows.
  3. Discard any 10 s window that still contains a NaN or is all zeros.

---

3. Downsampling (500 Hz → 50 Hz)

Your step ("downsample_to_50hz.py"):

```
python

# simply take every 10th sample (ppg_ds = ppg[::10]; abp_ds = abp[::10])
# new fs = 50
```

- You do no anti-alias filtering—just decimate by integer slicing.

Joung et al. approach:

- They assume (and briefly mention) that the original 500 Hz waveforms first pass through a low-pass/anti-alias filter in the monitor. After that, they downsample by a factor of 10 (500→50 Hz).
- For VitalDB, they even apply a **Savitzky–Golay smoothing** and **linear interpolation** before any downsampling (to remove high-frequency artifact) (see page 3–4 s41598-023-35492-y (6)).

Implications:

- Without any explicit anti-alias filtering, your simple `ppg[::10]` risks **aliasing**: high-frequency components (above 25 Hz) will fold into the 0–25 Hz band at 50 Hz. Although most physiologic PPG and ABP power sits below 15 Hz, any mechanical spike, motion artifact, or mains noise (often around 50 Hz or harmonics) will alias.
- The paper's "device-side" or "Savitzky–Golay" smoothing step removes much of that high-frequency content so that the downsampled result is faithful.
- **Recommendation:** Before subsampling, apply a zero-phase low-pass filter (e.g. Butterworth, cutoff ≈ 15 Hz for ABP; ≈ 8 Hz for PPG) or a short FIR filter to prevent aliasing. Then decimate by 10.

---

4. 10 s Window Splitting & SNR-Based Selection

Your step ("select_top_windows_with_snr")

- You take the 50 Hz signals and split them into nonoverlapping 10 s chunks (500 samples each).
- For each chunk you:
  1. Detect "raw peaks" in ABP and PPG using a local-max + minimum-distance (0.2 s) approach.
  2. Require at least 3 peaks in each 10 s window (so a floor of ~18 bpm).

3. Compute a simple SNR estimate:

$$\text{baseline} = \text{median}(\text{sig}), \quad \text{noise} = \text{MAD} = \text{median}(|sig - median(sig)|), \quad \text{signal\_amp}$$

$$\text{SNR} = \frac{\text{signal\_amp}}{\text{MAD} + \varepsilon}.$$

4. If SNR_ppg ≥ 2 and SNR_abp ≥ 2 and combined SNR ≤ 50, keep the chunk's score = SNR_ppg + SNR_abp.
5. Maintain a min-heap of size ≤ 300. At the end, pick the top 300 windows by combined SNR (sorted descending), then save their PPG/ABP pairs (plus `snr_scores` and `starts`).

Joung et al. approach (T4 + T5 filters):

1. They split into nonoverlapping 10 s at 50 Hz (exactly the same).
2. They do **not** compute or threshold on an SNR formula. Instead, for each window they enforce:
   - **T3:** Window must contain no NaNs or all-zero samples.
   - **T4:** Window's **average SBP** (from ABP) must lie in [70, 180] mmHg. If the mean SBP is outside that band, they drop the entire window.
   - They never talk about computing a "signal-to-noise ratio" or a "peak-count" filter. Their goal is simply to keep windows that are **physiologically plausible** (i.e. you can extract SBP/DBP) and have **no gross dropout**.
3. After T3+T4, they do **T5**: if a subject has fewer than 50 remaining windows, discard that subject completely; if it has > 100 windows, randomly select 100 to keep.

Implications:

- The paper's pipeline does **not** intentionally pick the "highest-SNR" windows; it picks windows within a plausible SBP range and with no missing data. They assume that if SBP∈[70,180] and there are no NaNs, the window is "good enough."
- Your SNR filter may be **too aggressive** in some cases (e.g. rejecting valid windows whose SNR happens to fall below 2.0) and **too lax** in others (e.g. accepting windows where ABP has a spurious artifact that meets your "peak + MAD" criteria but is not physiologic).
- Additionally, the paper's requirement that "average SBP must lie in [70,180]" automatically filters out any extreme hypotension or hypertension, which your SNR-only approach would not catch. If a window has mean SBP = 200 mmHg (artifact or real), you'd keep it if its SNR conditions were met; Joung et al. would throw it out immediately.
- **Recommendation:**
  1. Enforce a physiological SBP range (e.g. 70–180 mmHg) to replicate their T4 filter.
  2. Discard windows with any remaining NaNs or all-zero samples (your T3 analog).
  3. If you still want to use SNR as an additional "quality" check, treat it as a secondary filter—**not** the primary selection. For example, only compute SNR *after* T3/T4 pass and drop windows with very low SNR (< 2) (to remove flat-lines that sneak past the "≥ 3 peaks" rule). But do not base your "top N" on combined SNR.

---

5. SBP/DBP Label Extraction

Your step ("sbp_dbp_labels.py"):

- For each 10 s ABP window at 50 Hz:
  1. Detect systolic peaks using the same `find_peaks_with_min_distance` (prominence = 5 % of ABP range, min_dist = 0.3 s).
  2. If fewer than 2 peaks, set DBP = global ABP min; else find local minima (troughs) between consecutive detected peaks, then set DBP = median of those troughs.
  3. Set SBP = global ABP max.

Joung et al. approach:

- They compute SBP = maximum ABP in the window, DBP = minimum ABP in the window.
- They do not nuance "median of troughs." Their DBP is simply the global minimum amplitude in each 8.192 s window.
- (In VitalDB they smoothed with Savitzky–Golay first, then found peak vs trough; in UCI they simply took min and max, because the UCI data had already been filtered by Kachuee *et al.*.)

Implications:

- Your DBP method (median of troughs) is not "wrong"—it's arguably more robust to a single stray spike. But it is not what Joung et al. used, and if your trough-detector fails (e.g. misses a trough), you fallback to global min, which can sometimes pick a segment boundary artifact.
- If you want to compare your model's performance to theirs, you must use the same SBP/DBP definition (global max/min). Otherwise, any differences in DBP MAE might simply come from the label-extraction method.
- **Recommendation:** For consistency, compute DBP as the *global* minimum ABP sample (within each 10 s window) if you aim to replicate their results. If you keep "median-of-troughs," explicitly state that you deviated and justify why.

---

6. Normalization of SBP/DBP

Your pipeline:

- You did **not** explicitly standardize SBP and DBP to zero mean/unit variance. In the code you provided, after `sbp_dbp_labels.py` you simply save `(ppg_windows, sbp_values, dbp_values, fs)` for training.

Joung et al. approach:

- They take every SBP and DBP in the training set, compute $\mu_{\mathrm{SBP}}, \sigma_{\mathrm{SBP}}, \mu_{\mathrm{DBP}}, \sigma_{\mathrm{DBP}}$.
- Then for each training window they store

$$\mathrm{SBP}_{\mathrm{norm}} = \frac{\mathrm{SBP}_{\mathrm{raw}} - \mu_{\mathrm{SBP}}}{\sigma_{\mathrm{SBP}}}, \quad \mathrm{DBP}_{\mathrm{norm}} = \frac{\mathrm{DBP}_{\mathrm{raw}} - \mu_{\mathrm{DBP}}}{\sigma_{\mathrm{DBP}}}.$$

- At test time, they normalize with *those same* training μ/σ to avoid data leakage. The network's final outputs are on this standardized scale, then re-scaled to mmHg for reporting error.

Implications:

- If you feed "raw" SBP (say 65–150 mmHg) directly into your network, your loss landscape will be very different. The model may still learn something, but it will take longer to converge, and its weights will be biased toward larger-magnitude targets.
- Most implementations in the literature standardize their labels; this speeds up training and ensures SBP/DBP loss is balanced (1 mmHg error at 60 is "as important" as 1 mmHg at 140).
- **Recommendation:** Adopt the same normalization. After computing all SBP/DBP values for *all training windows*, calculate training-set mean/SD, and save them. Then, before training, convert every SBP_i and DBP_i to standardized units. Finally, at inference, convert network output back to mmHg.

---

7. Balancing Number of Windows per Subject

Your pipeline:

- After SNR-based selection, you simply pick the top 300 windows (by combined SNR) across the entire recording—there's no per-subject cap or minimum. If one subject's entire intraoperative case yielded 1,000 valid windows, you will keep all 300 (the top 300 SNR among those).
- If another subject only produced 20 valid windows, they would be entirely excluded (because your heap size is 300; you never pick them).

Joung et al. approach (T5):

- They require each subject to contribute **at least 50 windows**—if after T3+T4 a subject has fewer than 50, they drop the subject entirely (they never enter the train/val/test split).
- If a subject has more than 100 clean windows, they randomly sample 100 and throw away the rest.
- As a result, every subject in the final cohort has exactly 50–100 windows—no one subject can dominate the dataset.

Implications:

- Without balancing, your training set becomes heavily biased toward the single "best" subject (or a handful) who happened to have long, high-SNR recordings. The network may overfit to that subject's physiology (e.g. their heart rate range, blood-pressure range, waveform shape), and generalize poorly to others.
- Joung et al. explicitly use a subject-independent split: 2,987 subjects for training, 410 for validation, 797 for testing—and every subject has roughly the same number of samples. Their random-cap procedure prevents any one case from dominating the loss.
- **Recommendation:** Enforce a similar T5:
  1. After you finish SNR or T3/T4 filtering, count the remaining windows per subject.
  2. If a subject has < 50 windows, discard that subject entirely.

3. If a subject has > 100 windows, randomly pick 100 and discard the rest.
4. Now you can sample one window per subject per batch (or however you wish to compose batches), ensuring all subjects appear equally often.

What the Paper Leaves "Hidden" or Implicit

1. **Anti-Alias / Low-Pass Filtering Prior to Downsampling**
   - They mention "the device removes thermal noise" or "Savitzky–Golay smoothing" (VitalDB), but do not give explicit filter coefficients / orders. If you simply decimate, you'll alias.
   - **Hidden info to probe:** What exact low-pass filter (cutoff, order, zero-phase) did they use before subsampling 500 Hz→50 Hz? If you don't do something similar, your downsampled waveform may contain alias artifacts.
2. **Exact SBP/DBP Extraction Details**
   - In the VitalDB version, they "smooth by Savitzky–Golay, then take max/min" (p. 3–4 s41598-023-35492-y (6)). In UCI, they rely on Kachuee *et al.*'s preprocessing (which has already filtered the ABP).
   - **Hidden info to probe:** Which prominence and distance settings did they use for peak/trough detection? Did they ever exclude windows because "no peaks found" (or did they simply use global max/min)?
3. **How They Deal with Small NaN Gaps**
   - In VitalDB, they say: "All NaNs were interpolated linearly" before downsampling; but they do not say "if > X ms of NaN, we discard the window."
   - **Hidden info to probe:** If a PPG or ABP trace had, say, a 0.5 s dropout, did they still include that window (after interpolation)? Or did they drop it in T3? If you do only per-sample NaN removal, you lose synchronization.
4. **Train/Validation/Test Split Criteria**
   - They split by subject—but the exact random seed or stratification (e.g., by age or SBP distribution) is not fully detailed.
   - **Hidden info to probe:** Was the split completely random across all 4,185 subjects, or did they ensure a similar SBP/DBP distribution across train/val/test? If you simply randomly split, you could end up with a test set skewed toward very young/old or hypertensive/hypotensive extremes.
5. **Normalization of PPG and Derivatives**
   - They min–max normalize each PPG, VPG, and APG channel to [0, 1] (Eq. 3, p. 4 s41598-023-35492-y (6)), but they do not say whether they compute that normalization window-by-window or globally.
   - **Hidden info to probe:** Are those PPG/VPG/APG normalizations done per 8.192 s window (so each window's max → 1.0, min → 0.0), or do they compute min/max across the entire subject's recording? Or across all training data? Each choice can slightly shift distribution and affect convergence.

6. **Handling of Baseline Wander (High-Pass Filtering)**
    - They mention a "detrending" step (UCI) but do not describe its high-pass cutoff. In VitalDB, they apply SG smoothing.
    - **Hidden info to probe:** Did they apply a zero-phase high-pass (e.g. at 0.5 Hz) to the PPG before computing VPG/APG? That baseline removal can impact foot detection. If you skip it, your PPG foot might wander.
7. **Exact Window Overlap or Non-Overlap**
    - They segment into 8.192 s windows **with** 75 % overlap for UCI (p. 3 s41598-023-35492-y (6)). But in VitalDB, it is not explicitly stated whether they also use 75 % overlap or a simpler non-overlapping 10 s.
    - **Hidden info to probe:** Did they overlap windows on VitalDB? Overlap multiplies the number of training examples substantially. If you do nonoverlapping windows, you end up with fewer examples.

Is Your Pipeline "Good or Bad" Compared to the Paper?

- **Strengths of Your Pipeline:**
    1. You remove head/tail portions (30 min/10 min). If you have a strong rationale (e.g., the first 30 min always contain unstable anesthesia induction), this can reduce "edge-case" artifacts.
    2. You explicitly remove *all* NaNs so that no NaN remains within any window. (However, the paper prefers interpolation of small gaps plus discarding entire windows with NaNs rather than collapsing time.)
    3. You compute an explicit SNR measure to quantify window quality. That gives you a numeric "quality" metric beyond just "flatline vs non-flatline."
- **Weaknesses & Divergences:**
    1. **NaN removal by collapsing indices** misaligns PPG vs ABP timestamps, potentially breaking morphology. The paper instead interpolates or discards entire 10 s blocks with NaNs.
    2. **No anti-alias filter before downsampling.** Simply doing `x[::10]` can introduce aliasing. The paper's pipeline assumes at least some smoothing (device or Savitzky–Golay) before decimation.
    3. **SNR-based selection vs physiological SBP filter.** The paper does **not** pick "highest SNR" windows; they pick all windows whose average SBP $\in [70, 180]$ mmHg and that have no NaNs. Your SNR filter may accidentally accept artifact-ridden windows with high "peak amplitude" or discard valid low-SNR windows.
    4. **No SBP/DBP normalization.** Giving the raw mmHg values to a network is not far off, but it changes the scale of the loss. Standardizing helps training.
    5. **No per-subject balancing.** Without capping at 100 windows per subject or requiring a minimum of 50 windows, your network may see 90 % of its data from just a handful of "long" cases and none from short cases. Joung et al. go to great lengths to ensure each subject has at least 50 windows and at most 100, then split on subject.

- **Bottom-Line:**
  As written, your pipeline will produce valid PPG/ABP pairs in many cases, but it will **not** replicate the careful subject-and-window QC that Joung et al. used. You risk (a) corrupting time alignment when removing NaNs, (b) aliasing when downsampling, (c) training heavily on a small subset of "high-SNR" subjects, and (d) having scale-mismatch between your SBP/DBP labels and theirs. If your goal is simply "get 100 plausible strips," you might survive—but if you want to reimplement their method faithfully (or compare against their reported MAEs), you'll need to incorporate their T1–T5 steps (demographic filtering, segmentation with no overlap, physiological SBP range, per-subject window capping) and label-normalization procedures.

Which "Hidden" Details You Should Consider Adding

1. **Anti-Alias Filtering before Decimation:**
   - Insert a zero-phase low-pass (e.g. 4th-order Butterworth at 15 Hz) on ABP and a separate low-pass (8 Hz) on PPG (or use Savitzky–Golay smoothing) before taking every 10th sample.
   - This prevents aliasing of motion spikes or mains interference into the 0–25 Hz band after downsampling.
2. **Physiological SBP Range Filter (T4):**
   - Rather than relying solely on SNR, compute each window's average or median SBP. Discard any window whose mean SBP $\notin$ [70, 180] mmHg. This is the simplest way to remove out-of-range artifacts.
3. **NaN Handling by Window-Level Drop, Not Sample-Level Collapse:**
   - Option A: Linearly interpolate short (< 200 ms) NaN gaps before any downsampling.
   - Option B: If a downsampled 10 s window still contains any NaNs, drop that entire window (T3). Do not collapse samples, so your windows remain exactly 500 samples long.
4. **SBP/DBP Normalization:**
   - After extracting SBP_i (global max) and DBP_i (global min), compute training-set μ/σ for each, and store "SBP_norm_i" = (SBP_i − μ_SBP)/σ_SBP, similarly for DBP. Train on those standardized labels.
   - At inference, re-scale back to mmHg for error reporting.
5. **Subject-Level Window Balancing (T5):**
   - After you've applied T3+T4 (window-level QC), count how many valid windows each subject has. If a subject has fewer than 50, drop that subject entirely. If they have > 100, randomly pick 100. This ensures approximate subject-independence and prevents a few long anesthetic cases from dominating your loss.
6. **Exact Window Overlap/Stride:**
   - Joung et al. use **no overlap** for VitalDB, but for UCI (125 Hz) they use 75 % overlap. If you want more training data, decide whether you will allow some fraction of overlap. If you do overlap, be explicit about your stride length (e.g. 500 samples window with 250 samples overlap).
7. **Split Strategy & Label Leakage:**

- Always normalize SBP/DBP with the training set's statistics only.
- Choose a fixed random seed to split subjects into train/val/test so results are reproducible.
- Make sure that **no window** from a given subject ever appears in more than one split.

8. **Potentially Missing Demographic Filters (T1):**
    - The paper drops pediatric (< 18 years), extreme weight/height values, and pregnant patients. If your dataset includes such outliers, consider adding a similar demographic filter. Otherwise, you may be training on physiology the paper specifically excluded.

In Summation

- **If your goal is to replicate Joung et al.'s results as closely as possible**, you'll need to:
    1. **Drop entire windows** containing NaNs (rather than collapsing indices).
    2. **Apply a true anti-alias filter** before decimation (or use the existing device's anti-alias).
    3. **Filter windows by mean SBP ∈ [70,180] mmHg** (instead of an arbitrary SNR cutoff).
    4. **Normalize SBP/DBP** to zero mean/unit variance (training-set statistics).
    5. **Cap each subject to ≥ 50 and ≤ 100 windows** before training/validation/test splits.
- **If you only care about getting "some clean strips"** for a quick proof of concept, you can stick with your SNR-based selection, but be aware that:
    - Your windows will be misaligned if you collapse NaN indices.
    - You risk aliasing if you decimate without filtering.
    - Your highest-SNR windows may still contain large artifacts (in fact, you saw that scenario).
    - You may overfit to a small handful of subjects who happened to have long, high-SNR recordings.

By adding the missing pieces (anti-alias filter, SBP-range filter, per-subject capping, label normalization), you will more closely match the paper's rigor—and avoid subtle sources of bias or data-leakage that could artificially inflate your performance.

ok to replicate the architecture as in the paper how shd i structure my final data and in what format shd i store per patient ,

To train the U²-Net + BiLSTM architecture exactly as Pan et al. did, you need to prepare, for each patient (i.e. each "case"), a fixed collection of synchronized, 10 s windows at 50 Hz of:

1. **Input channels**:
   - The raw (detrended, smoothed) PPG waveform,
   - Its first derivative (VPG),
   - Its second derivative (APG).
2. **Target channel**:
   - The corresponding ABP waveform (at 50 Hz).
3. **Scalar labels** (if you also want to report SBP/DBP errors):
   - SBP = (max value of ABP in that 10 s window),
   - DBP = (min value of ABP in that 10 s window).

Below is a recommended data layout (per patient) that mirrors exactly what the paper expects at training time. You can store each patient's data as a single compressed NumPy bundle (`.npz`), with the following arrays:

```
sql

patient_<ID>.npz
├── ppg_windows      shape = (N_windows, 500)    # raw PPG at 50 Hz (10 s each)
├── vpg_windows      shape = (N_windows, 500)    # first derivative of ppg_windows
├── apg_windows      shape = (N_windows, 500)    # second derivative of ppg_windows
├── abp_windows      shape = (N_windows, 500)    # corresponding ABP at 50 Hz
├── sbp_labels       shape = (N_windows,)        # SBP (mmHg) per window
├── dbp_labels       shape = (N_windows,)        # DBP (mmHg) per window
└── fs               float scalar (should be 50.0)
```

where:

- **N_windows** is the number of 10 s windows you kept for that patient (after QC).
  - Per Pan et al., each patient should contribute **50–100 windows**.
  - If a patient originally has fewer than 50 valid windows, discard that patient; if they have more than 100, randomly select exactly 100 to keep.
- **"500"** is because each 10 s window at 50 Hz has 10 s × 50 samples/sec = 500 samples.
- **ppg_windows[i, :]** is the i-th 10 s clip of the (low-passed + detrended) PPG signal at 50 Hz.
- **vpg_windows[i, :]** = finite difference of ppg_windows[i], e.g.

```
python

vpg_windows[i] = ppg_windows[i, 1:] – ppg_windows[i, :-1]
# (you can pad one value at the front or back so it's also length 500)
```

- **apg_windows[i, :]** = finite difference of vpg_windows[i], again padded to length 500.
  This matches Eq. 1–2 in Pan et al. (2024).
- **abp_windows[i, :]** is the target ABP waveform (50 Hz) for that same 10 s.
  Before slicing into windows, you should have:
  1. Smoothed ABP (e.g. Savitzky–Golay) or a zero-phase low-pass at ~15 Hz (VitalDB),
  2. No high-pass on ABP (so you preserve the true baseline and absolute mmHg values).
- **sbp_labels[i]** = `np.max(abp_windows[i, :])` (the true systolic peak in mmHg).
- **dbp_labels[i]** = `np.min(abp_windows[i, :])` (the true diastolic trough in mmHg).
  (Pan et al. simply took global max/min in each 8.192 s window; since you're using 10 s windows, do the same.)
- **fs** = 50.0 (a float), so the model knows the sampling rate.

Step-by-Step Recipe to Build Each `patient_<ID>.npz`

Below is a condensed workflow (you can script this in Python) to go from your raw 500 Hz files to exactly this per-patient layout. In each bullet, you'll see how it corresponds to Pan et al.'s T1–T5 steps:

1. **Case-Level Filtering (T1 + T2)**
   - (Pan et al. first dropped any patient with invalid demographics or missing PPG/ABP. If you've already done that, skip this part; otherwise, check age/weight/height/pregnancy, and skip patients lacking full synchronous PPG+ABP traces.)
2. **Strip Unstable Periods**
   - You mentioned stripping the first 30 min and last 10 min. If you truly want to replicate Pan et al., they do **not** remove fixed head/tail portions. Instead, they keep all data, trusting that any bad 10 s windows will be dropped in T3/T4.
   - If you prefer to preserve exactly what they did, comment out (or skip) your "remove first 30 min / last 10 min" step. They never did that.
3. **Interpolate Small NaN Gaps**
   - **VitalDB**: Pan et al. linearly interpolated any NaNs *before* downsampling, as long as the gap was short.
   - **UCI**: their inputs were already prefiltered, so there were no NaNs.
   - **Your code currently** collapses NaN-samples outright. Instead, do:

     ```
     python

     # if either ppg[i] or abp[i] is NaN but the gap < ~0.2 s (10 samples at 50 Hz),
     # interpolate linearly between the two valid neighbors.
     # If the gap > ~0.2 s, mark that entire 10 s window as invalid and drop it later.
     ```

- In practice:

```
python

import numpy as np
def fill_short_nans(sig, max_gap_samples=10):
    # find runs of NaNs; if run length <= max_gap_samples, interpolate
    # else leave them as NaN for T3 to drop.
    isnan = np.isnan(sig)
    if not np.any(isnan):
        return sig
    cleaned = sig.copy()
    # find start/end indices of each NaN run
    idx = 0
    N = len(sig)
    while idx < N:
        if not isnan[idx]:
            idx += 1
            continue
        # run of NaNs starts here
        run_start = idx
        while idx < N and isnan[idx]:
            idx += 1
        run_end = idx  # first index after NaN run
        run_len = run_end - run_start
        if run_len <= max_gap_samples and run_start > 0 and run_end < N:
            # interpolate between cleaned[run_start - 1] and cleaned[run_end]
            y0 = cleaned[run_start - 1]
            y1 = cleaned[run_end]
            for k in range(run_len):
                alpha = (k + 1) / (run_len + 1)
                cleaned[run_start + k] = (1 - alpha) * y0 + alpha * y1
    return cleaned
```

- After this step, you'll have **`ppg_interp`** and **`abp_interp`** at 500 Hz, with only very short gaps filled. Any longer gaps remain NaN.

4. **Anti-Alias / Low-Pass Filtering**

- Before you downsample, apply a zero-phase low-pass to both PPG and ABP. For example:

```
python

from scipy.signal import butter, filtfilt

def lowpass_filter(sig, fs, cutoff=15.0, order=4):
    # zero-phase 4th-order Butterworth @ cutoff Hz
    b, a = butter(order, cutoff/(fs/2), btype='lowpass')
    return filtfilt(b, a, sig)

# apply to ppg_interp and abp_interp
ppg_smooth500 = lowpass_filter(ppg_interp, fs=500, cutoff=8.0)  # PPG band ~ < 8 Hz
abp_smooth500 = lowpass_filter(abp_interp, fs=500, cutoff=15.0)  # ABP band ~ < 15 Hz
```

- This replicates their "device smoothing" or "SG smoothing" step. Without this, aliasing will occur when you slice.

5. **Downsample to 50 Hz**

- Now safely take every 10th sample:

```
python

ppg_50 = ppg_smooth500[::10]   # length → round(len/10)
abp_50 = abp_smooth500[::10]
fs50 = 50.0
```

- At this point, any NaNs that remained from step 3 (gaps longer than 0.2 s) are still NaN in `ppg_50` or `abp_50`. You will catch them in T3.

6. **Form Non-Overlapping 10 s Windows**
   - Let `window_len = int(fs50 * 10) # 500 samples`.
   - Compute `num_windows = len(ppg_50) // 500`.
   - For each `i` from 0 to `num_windows – 1`, define:

```python
start = i * 500
end   = start + 500
ppg_win_i = ppg_50[start:end]   # shape (500,)
abp_win_i = abp_50[start:end]   # shape (500,)
```

   - **Pan et al. did not overlap windows** (VitalDB). If you wanted more data, you could use overlap; but to replicate their exact setup, use no overlap.

7. **Window-Level QC (T3 + T4)**
   - **T3 (No NaNs / No All-Zerofields)**

```python
if np.any(np.isnan(ppg_win_i)) or np.any(np.isnan(abp_win_i)):
    # discard this window entirely
    continue
if np.all(ppg_win_i == 0) or np.all(abp_win_i == 0):
    # flatline → discard
    continue
```

   - **T4 (Mean SBP ∈ [70, 180] mmHg)**

```python
sbp_i = float(np.max(abp_win_i))
# optionally take mean of all local peaks instead of global max. But Pan et al. used global
if sbp_i <  70 or sbp_i > 180:
    continue
```

   - If the window passes both T3 and T4, collect it in temporary lists:

```python
valid_ppg_wins.append(ppg_win_i)
valid_abp_wins.append(abp_win_i)
valid_sbp_vals.append(sbp_i)
valid_dbp_vals.append(float(np.min(abp_win_i)))
# you'll compute VPG/APG next
```

8. **Compute VPG and APG for Each Window**

Uploaded image
i am taking abt thos arhcitectur e

Below is a detailed walk-through of the PPG2BP-Net architecture as shown in Joung et al. (2023), with an emphasis on how the various blocks (1D-CNN, MLP calibration, fusion, and final output layers) are connected. This should clarify exactly what "goes where" when you implement their model.

---

1. Overall High-Level Structure

PPG2BP-Net is designed to predict a patient's **instantaneous SBP and DBP** (two scalar outputs) for a new ("target") 10 s PPG strip, **given** a short "calibration" PPG strip plus its known SBP/DBP. In other words, you give the network:

1. **Target PPG**: a 10 s window (500 samples at 50 Hz) for which you want to estimate SBP/DBP.
2. **Calibration PPG**: a (typically shorter) PPG segment for which you *already know* SBP and DBP (from an invasive cuff or transducer).
3. **Calibration SBP & DBP**: the two scalar values (e.g. 120 mmHg and 75 mmHg) measured concurrently with the calibration PPG.

The model then "personalizes" its mapping from PPG→BP by comparing the target PPG features with the calibration PPG features, and by taking into account the calibration SBP/DBP. In practice, the network has three main sub-networks:

1. **Shared 1D-CNN (in red)** that embeds any PPG chunk (target or calibration) into a 1D "feature vector."
2. **Calibration MLP (in blue)** that embeds the two scalar calibration labels (SBP_cal, DBP_cal) into a learned "calibration embedding."
3. **Fusion & Output MLP (in purple)** that concatenates the absolute-difference of CNN features (target vs calibration) with the calibration embedding, and finally outputs predicted SBP/DBP for the target window.

Below is a block-by-block description.

---

2. The Shared 1D-CNN ("Red" Branch)

This is the backbone that extracts a fixed-size feature vector from any PPG waveform. It is used *twice* – once on the target PPG window and once on the calibration PPG window. In code you'll literally create one `tf.keras` or `torch.nn` module for this CNN, then call it on both inputs (sharing weights).

2.1 Input

- **Shape**: `(batch_size, L)`, where `L = 500` samples (i.e. 10 s × 50 Hz).
  In practice, you might add a "channel" dimension so the Tensor shape is `(batch_size, 1, 500)` for a

1D-CNN.

2.2 Convolutional Layers

The paper's Fig. 4 (red) shows **five convolutional blocks** ("Conv Layer 1" through "Conv Layer 5"). Each block consists of:

1. **1D Convolution**
   - Filter width: 3 samples (kernel size = 3)
   - Number of filters: increases as you go deeper (e.g. 16→32→64→128→256).
   - Stride: 1 (no temporal down-sampling here).
2. **Batch Normalization**
   - Applies BN along the channel dimension after each convolution.
3. **ReLU Activation**
   - Nonlinear rectifier.
4. **(Between Block 4 and Block 5)** there is **one Average-Pooling** operation. Typically this is a 1D average-pool with pool size = 2 (halves the temporal dimension). The schematic shows "Avg pooling" after Conv Layer 4 before going into Conv 5.
5. **Dropout**
   - After the last Conv + BN + ReLU, they insert a Dropout (rate ≈ 0.5) to prevent overfitting.
6. **Fully Connected "FCL" at the very end** (white box after Conv 5 in Fig. 4)
   - Following Conv 5 → BN → ReLU, they flatten the feature maps and feed them into:
     - One Dense (fully connected) layer
     - Then BN + ReLU again
   - The result is a **1D feature vector** of size D (e.g. 128 or 256, depending on how they set it). This feature vector is the "embedding" of whatever PPG window you passed in.

Rough Data-Flow in the 1D-CNN

- **Input**: shape `(batch, 1, 500)`
  → Conv1D(filters=16, kernel=3, stride=1) → BN → ReLU →
  → Conv1D(filters=32, kernel=3, stride=1) → BN → ReLU →
  → Conv1D(filters=64, kernel=3, stride=1) → BN → ReLU →
  → Conv1D(filters=128, kernel=3, stride=1) → BN → ReLU → **AvgPool1D(pool_size=2)** (now temporal length is 250)
  → Conv1D(filters=256, kernel=3, stride=1) → BN → ReLU → Dropout(0.5) →
  → Flatten → Dense(D) → BN → ReLU → ***output***: shape `(batch, D)`.

  Important: Exactly how many filters and what pooling sizes they used is gleaned from the figure's brick-sizes and the text ("Five 1D-CNN layers, with batch-norm+ReLU after each, an average-pool after layer 4, and a final FCL of size 128"). If you want to be precise, set:

```python
conv_filters = [16, 32, 64, 128, 256]
kernel_size  = 3
pool_after_layer = 4
final_FC_size = 128  # so output embedding is 128-dimensional
```

You will re-use *exactly this same 1D-CNN* (same weights) on:

- **"Target PPG"** → yields feature vector $\mathbf{f}_{\text{tgt}} \in \mathbb{R}^D$.
- **"Calibration PPG"** → yields feature vector $\mathbf{f}_{\text{cal}} \in \mathbb{R}^D$.

---

3. Calibration MLP ("Blue" Branch)

Parallel to the CNN, the architecture uses a small MLP (multi-layer perceptron) that takes in the **two scalar labels** from the calibration window—(SBP_cal, DBP_cal)—and maps them into another learned embedding. In Fig. 4 you'll see two identical "towers" (left and right) in the blue block: one for SBP_cal and one for DBP_cal. Then those two outputs are concatenated.

3.1 Inputs

- **SBP_cal**: a scalar (e.g. 120.0 mmHg), typically already *normalized* (zero-mean, unit-variance) across the training set.
- **DBP_cal**: a scalar (e.g. 75.0 mmHg), also normalized.

You feed each of those scalars into its own mini-MLP (but they share the same architecture/weights). Concretely:

3.2 MLP Structure (for one scalar)

1. **FCL 1 (Dense)**
   - Input: 1-dimensional scalar.
   - Output: 64 units.
   - Followed by Batch-Norm → ReLU.
2. **FCL 2 (Dense)**
   - Input: 64 units.
   - Output: 32 units.
   - Followed by Batch-Norm → ReLU.

After that, you have a 32-dimensional embedding for SBP_cal, call it $\mathbf{e}_{\text{SBP}} \in \mathbb{R}^{32}$. Similarly you run DBP_cal through the *same* two-layer MLP (same weights) to get $\mathbf{e}_{\text{DBP}} \in \mathbb{R}^{32}$.

3.3 Concatenation

- You then concatenate those two embeddings:

$$\mathbf{e}_{\text{cal}} = \left[\mathbf{e}_{\text{SBP}} \,\|\, \mathbf{e}_{\text{DBP}}\right] \in \mathbb{R}^{64} \quad (\text{just stacking the 32-dim SBP embed} + \text{32-dim DBP embed})$$

That $\mathbf{e}_{\text{cal}}$ is your final "calibration embedding" of dimension 64.

---

4. Fusion and Final Output ("Purple" Branch)

Now you have:

1. $\mathbf{f}_{\text{tgt}} \in \mathbb{R}^D$: the 1D-CNN embedding of the target 10 s PPG.
2. $\mathbf{f}_{\text{cal}} \in \mathbb{R}^D$: the 1D-CNN embedding of the calibration PPG.
3. $\mathbf{e}_{\text{cal}} \in \mathbb{R}^{64}$: the MLP embedding of (SBP_cal, DBP_cal).

The idea is that the **difference** between $\mathbf{f}_{\text{tgt}}$ and $\mathbf{f}_{\text{cal}}$ captures how the target waveform differs from the calibration waveform in the learned feature space; meanwhile, $\mathbf{e}_{\text{cal}}$ tells the network what the "true" BP was at calibration time. Together, they let the model "adjust" the known calibration BP to predict the unknown target BP.

4.1 Absolute-Difference of CNN Features

Compute a vector of length D:

$$\Delta\mathbf{f} = \left|\mathbf{f}_{\text{tgt}} - \mathbf{f}_{\text{cal}}\right|,$$

where the absolute value is taken elementwise. In code, this is just:

```
python

delta_f = torch.abs( f_tgt - f_cal )   # shape = (batch_size, D)
```

4.2 Concatenate with Calibration Embedding

Now stack $\Delta\mathbf{f} \in \mathbb{R}^D$ with $\mathbf{e}_{\text{cal}} \in \mathbb{R}^{64}$. The resulting vector has dimension $D + 64$. For example, if your CNN's final embedding size $D = 128$, then

- $\Delta\mathbf{f} \in \mathbb{R}^{128}$,
- $\mathbf{e}_{\text{cal}} \in \mathbb{R}^{64}$,
- Total concatenated feature $\mathbf{h} = \left[\,\Delta\mathbf{f} \,\|\, \mathbf{e}_{\text{cal}}\,\right] \in \mathbb{R}^{192}$.

4.3 Final Dense Layers to SBP/DBP

That fused vector $\mathbf{h}$ is now fed into a small MLP (the purple block in Fig. 4):

1. **FCL 1**
   - Input dimension: $D + 64$.
   - Output dimension: 128 units.
   - Followed by BatchNorm $\rightarrow$ ReLU.

2. **FCL 2**
    - Input dimension: 128.
    - Output dimension: 2 units (these will be the final $\widehat{\mathrm{SBP}}$ and $\widehat{\mathrm{DBP}}$).

Thus the final output of the entire network is a 2-dimensional vector $[\widehat{\mathrm{SBP}}, \widehat{\mathrm{DBP}}]$. Because SBP/DBP were *normalized* at training time, the network actually outputs "normalized" SBP & DBP; you then invert the normalization (multiply by σ, add μ) to get mmHg.

---

5. Full Data Flow (Step by Step, for One Minibatch)

Below is a concrete pseudo-code outline of how one forward-pass would look. Suppose your minibatch size is B. You prepare:

- **`target_ppg`** shape = **`(B, 1, 500)`**
- **`cal_ppg`** shape = **`(B, 1, 500)`**
- **`cal_sbp`** shape = **`(B,)`** (already normalized)
- **`cal_dbp`** shape = **`(B,)`** (already normalized)

Then:

```
python

# 1) Shared 1D-CNN on both PPG inputs:
f_tgt = CNN1D( target_ppg )    # shape = (B, D)
f_cal = CNN1D( cal_ppg )       # shape = (B, D)
#   CNN1D = the five-layer conv → pool → conv → flatten → dense → BN → ReLU

# 2) Compute element-wise absolute difference:
delta_f = torch.abs( f_tgt - f_cal )    # shape = (B, D)

# 3) Calibration MLP on SBP and DBP (two parallel branches, same weights):
e_sbp = MLPcal( cal_sbp.unsqueeze(-1) ) # shape = (B, 32)
e_dbp = MLPcal( cal_dbp.unsqueeze(-1) ) # shape = (B, 32)
#   MLPcal = Dense(1→64) → BN → ReLU → Dense(64→32) → BN → ReLU

# 4) Concatenate SBP/DBP embeddings:
e_cal = torch.cat([e_sbp, e_dbp], dim=-1)  # shape = (B, 64)

# 5) Concatenate delta_f with e_cal:
h = torch.cat([ delta_f, e_cal ], dim=-1)   # shape = (B, D+64)

# 6) Final MLP:
x = Dense( (D+64)→128 )( h )    # → BN → ReLU
out = Dense( 128 → 2 )( x )     # no activation (these are raw pred SBP_norm, DBP_norm)

# 7) At training time, compute loss L between out[:,0] (pred_norm_SBP) vs (true_norm_SBP)
# and out[:,1] (pred_norm_DBP) vs (true_norm_DBP). Typically use MSE or MAE, etc.

# 8) At inference, do:
pred_SBP_mmHg = out[:,0] * sigma_SBP + mu_SBP
pred_DBP_mmHg = out[:,1] * sigma_DBP + mu_DBP
```

If **`D=128`** (embedding from CNN1D), then the final concatenated vector has size **`128 + 64 = 192`**.

---

6. What Weight Sharing & Calibration Means

- Notice how the same CNN (`CNN1D`) is used on both the target PPG and the calibration PPG. This weight-sharing ensures that both inputs are embedded into the *same* feature space.
- The **absolute-difference** $|f_{\text{tgt}} - f_{\text{cal}}|$ then captures how the new PPG differs morphologically from the calibration PPG. If the patient's physiology has shifted (e.g. their pulse-transit time changed because of a blood-pressure drift), the difference vector will capture that shift.
- Meanwhile, the **MLPcal** embedding of `(SBP_cal, DBP_cal)` tells the network "the last time we measured exactly SBP_cal, DBP_cal, here's how the PPG looked (via f_cal)." The model is effectively learning:

$$(\text{current SBP}, \text{DBP}) = (\text{cal SBP}, \text{DBP}) \; + \; \text{some function of} \left|\mathbf{f}_{\text{tgt}} - \mathbf{f}_{\text{cal}}\right|.$$

  The final two-layer MLP (purple) is exactly modeling that "increment."

---

7. Putting It Into Your Code / Data

7.1 Data You Must Supply, Per Patient

For each patient, you should prepare:

1. **Calibration data** (picked once per patient or periodically):
   - A short "calibration PPG" window, e.g. 10 s of PPG at 50 Hz, stored as an array of shape `(500,)`.
   - The corresponding `(SBP_cal, DBP_cal)` recorded simultaneously.
   - (In practice, Joung et al. used a single 10 s calibration block per subject before each surgery; you could instead sample multiple calibration windows over time—whatever your experimental design calls for.)
2. **Many target windows**:
   - A set of 10 s PPG strips at 50 Hz, each shape `(500,)`, all drawn from later in that same patient's recording.
   - During training, you'll pair each target PPG window with the same calibration PPG+labels (because it's subject-specific).
   - If you want to mimic their training/validation split, ensure that all windows for one subject go together into train, valid, or test (never split a subject across sets).
3. **On-Disk Format** (per patient, save as a single `.npz` file):

```
text

patient_XYZ.npz
├── cal_ppg      shape = (500,)  # the 10 s calibration PPG
├── cal_sbp      scalar (float)  # the SBP in mmHg for cal_ppg
├── cal_dbp      scalar (float)  # the DBP in mmHg for cal_ppg
├── tgt_ppg      shape = (N, 500) # N target windows, each 10 s = 500 samples
├── tgt_sbp      shape = (N,)     # "ground-truth" SBP for each of those N windows
├── tgt_dbp      shape = (N,)     # "ground-truth" DBP for each of those N windows
└── fs           scalar (50.0)
```

   - Here, `N` might be 50–100 or however many valid windows you extracted *after* T3–T5 filtering.

- You will later load that `.npz` and, for each minibatch, pick one row from `tgt_ppg` (plus its label) to form your training examples.

7.2 How to Use That File in Training

In your PyTorch/TensorFlow dataloader:

1. **Load** `patient_XYZ.npz`.
2. **Normalize**:
   - `ppg` is already downsampled + filtered → use as-is (you may min-max normalize or z-score each 500-sample window to [0,1], depending on the paper's exact choice).
   - `cal_sbp` and `cal_dbp`: apply the training-set μ/σ to convert to "normalized SBP/DBP" and store those in memory.
   - `tgt_sbp`, `tgt_dbp`: also normalize with the *same* μ/σ.
3. **At each training step**:
   - Pick a random index `i` from `[0..N-1]` for that patient.
   - Form:

```python
target_ppg = tgt_ppg[i, :]     # shape (500,)
sbp_true   = tgt_sbp[i]        # normalized scalar
dbp_true   = tgt_dbp[i]        # normalized scalar
```

   - Your "calibration input" is always (`cal_ppg`, `cal_sbp_norm`, `cal_dbp_norm`)—the same for every `i` in that patient.
   - **Stack** these per-patient examples across a minibatch of `B` different patients.
   - **Forward-pass** as shown in Section 5 above, compute `pred_sbp_norm, pred_dbp_norm`.
   - **Compute loss** = `L1( pred_sbp_norm, sbp_true ) + L1( pred_dbp_norm, dbp_true )` (or some weighted combination, as the paper states).
   - **Backpropagate** and update weights (CNN1D + MLPcal + final MLP).

At **inference** time you do exactly the same except you do not have `tgt_sbp, tgt_dbp`; you only do:

- `pred_norm_sbp, pred_norm_dbp = model( target_ppg, cal_ppg, cal_sbp_norm, cal_dbp_norm )`
- Then convert back:

```python
pred_sbp_mmHg = pred_norm_sbp * sigma_SBP + mu_SBP
pred_dbp_mmHg = pred_norm_dbp * sigma_DBP + mu_DBP
```

---

8. Summary of Key Points to Make Your Implementation "Faithful"

1. **Data structure per patient**:
   - One `.npz` (or HDF5) containing exactly:
     - `cal_ppg` (500 samples)
     - `cal_sbp` (scalar), `cal_dbp` (scalar)
     - `tgt_ppg` (shape `(N,500)`)
     - `tgt_sbp` (shape `(N,)`), `tgt_dbp` (shape `(N,)`)
     - `fs = 50.0`
2. **Network Building Blocks**:
   - **1D CNN** (shared) that maps 500 samples → embedding of size 128 (for instance). Five convolutional blocks, batch-norm, ReLU, average-pool after block 4, dropout, then one Dense→BN→ReLU.
   - **Calibration MLP**: Two identical sub-MLPs that each map one scalar→32, then concatenated to a 64-dim vector.
   - **Fusion MLP**: Concatenate $|f_{\text{tgt}} - f_{\text{cal}}|$ (128-dim) with the 64-dim calibration embedding → 192-dim → Dense(192→128)→BN→ReLU → Dense(128→2) output.
3. **Loss & Normalization**:
   - Before training, compute μ_{SBP}, σ_{SBP} (and similarly for DBP) *only* on the training set's SBP/DBP.
   - Use those to convert each "true" SBP_i into `sbp_norm_i = (sbp_i–μ_SBP)/σ_SBP`. Do the same for DBP.
   - Train on those normalized labels; at inference, invert the scaling.
4. **Training Loops & Subject Splitting**:
   - Ensure each patient's windows and calibration pair appear only in one of train/val/test splits.
   - Within a minibatch, you should sample exactly one target-window index per patient (so each batch is "B patients × 1 window each"). This matches the paper's "one sample + one calibration" per patient per iteration.
   - If a patient has multiple calibration windows (e.g. one at induction, one midway), you can experiment with either: (a) always using the same first-window calibration, or (b) randomly picking one calibration for each new target. The paper used a **single** calibration window per subject.
5. **Implementation Tips**:
   - Use **zero-padding** on the Conv1D layers so every convolution keeps the same temporal length (500 → 500, until you do the average-pool).
   - Make sure your average-pool filter exactly halves the length (so that after 4 conv blocks + pool, the temporal dimension is $500/2 = 250$).
   - Keep careful track of dimensions: after the final Conv1D+ReLU, you'll have a tensor shape `(batch, 256, 250)` (if your 5th convolution has 256 filters). Flatten that to `(batch, 256×250 = 64 000)` before the final Dense → 128.
   - Alternatively, you can insert extra pooling layers or reduce filter counts (e.g. 128 → 64) to shrink the flatten size, but then you won't be exactly what the paper used. The figure's proportions

imply a 1×1 conv or a global-average-pool before the final Dense, but their text says "flatten → Dense(128)." If memory is an issue, you can do a GlobalAveragePool1D after Conv5 to reduce the feature map to `(batch, 256)`, then Dense→128. That's slightly different, but more efficient.

Example PyTorch-style Pseudocode

Below is a condensed PyTorch-like sketch (omitting full import statements and activation details). You can adapt it to TensorFlow if you prefer.

```
python

import torch
import torch.nn as nn
import torch.nn.functional as F

class PPG2BPNet(nn.Module):
    def __init__(self, fs=50):
        super().__init__()
        # ----------------------------------
        # 1) 1D CNN Backbone (shared)
        # ----------------------------------
        self.conv1 = nn.Conv1d(1,  16, kernel_size=3, padding=1)
        self.bn1   = nn.BatchNorm1d(16)

        self.conv2 = nn.Conv1d(16, 32, kernel_size=3, padding=1)
        self.bn2   = nn.BatchNorm1d(32)

        self.conv3 = nn.Conv1d(32, 64, kernel_size=3, padding=1)
        self.bn3   = nn.BatchNorm1d(64)

        self.conv4 = nn.Conv1d(64,128, kernel_size=3, padding=1)
        self.bn4   = nn.BatchNorm1d(128)

        # After conv4, we do avg-pool (kernel_size=2) → length 250 from 500
        # Then conv5:
        self.conv5 = nn.Conv1d(128,256, kernel_size=3, padding=1)
        self.bn5   = nn.BatchNorm1d(256)
        self.drop5 = nn.Dropout(p=0.5)

        # Compute final flatten dimension: 256 channels × 250 timepoints = 64 000
        # Dense to 128
        self.fc_cnn = nn.Linear(256*250, 128)
        self.bn_fc  = nn.BatchNorm1d(128)

        # ----------------------------------
        # 2) Calibration MLP (shared for SBP & DBP)
        # ----------------------------------
        self.fc_cal1 = nn.Linear(1, 64)
        self.bn_cal1 = nn.BatchNorm1d(64)

        self.fc_cal2 = nn.Linear(64, 32)
        self.bn_cal2 = nn.BatchNorm1d(32)

        # ----------------------------------
        # 3) Final Fusion MLP
        # ----------------------------------
        # Input dimension = (128 from |f_tgt-f_cal|) + (32+32 from cal SBP/DBP) = 192
        self.fc_fuse1 = nn.Linear(128 + 64, 128)  # 128 + 64 = 192
        self.bn_fuse1 = nn.BatchNorm1d(128)
        self.fc_fuse2 = nn.Linear(128, 2)         # outputs (SBP_norm, DBP_norm)

    def cnn_embed(self, x_ppg):
        """
        Shared 1D-CNN that maps input PPG (1×500) → embedding of size 128.
        x_ppg: shape (B, 1, 500)
        """
        x = F.relu(self.bn1(self.conv1(x_ppg)))   # → (B, 16, 500)
        x = F.relu(self.bn2(self.conv2(x)))       # → (B, 32, 500)
```

```
        x = F.relu(self.bn3(self.conv3(x)))       # → (B, 64, 500)
        x = F.relu(self.bn4(self.conv4(x)))       # → (B,128, 500)
        x = F.avg_pool1d(x, kernel_size=2)        # → (B,128, 250)
        x = F.relu(self.bn5(self.conv5(x)))       # → (B,256, 250)
        x = self.drop5(x)                         # → (B,256, 250)
        x = x.view(x.size(0), -1)                 # → (B, 256*250 = 64000)
        x = F.relu(self.bn_fc(self.fc_cnn(x)))    # → (B, 128)
        return x                                  # this is f_ppg

    def cal_embed(self, cal_scalar):
        """
        MLP that maps one scalar (SBP or DBP) → embedding of size 32.
        cal_scalar: shape (B,) i.e. (B,) but we reshape to (B,1)
        """
        x = cal_scalar.unsqueeze(-1)              # → (B,1)
        x = F.relu(self.bn_cal1(self.fc_cal1(x)))  # → (B,64)
        x = F.relu(self.bn_cal2(self.fc_cal2(x)))  # → (B,32)
        return x                                  # embedding of SBP or DBP

    def forward(self, tgt_ppg, cal_ppg, cal_sbp, cal_dbp):
        # 1) shared CNN embeddings
        f_tgt = self.cnn_embed(tgt_ppg)            # → (B,128)
        f_cal = self.cnn_embed(cal_ppg)            # → (B,128)

        # 2) difference
        delta_f = torch.abs(f_tgt - f_cal)         # → (B,128)

        # 3) parameter embeddings
        e_sbp = self.cal_embed(cal_sbp)            # → (B,32)
        e_dbp = self.cal_embed(cal_dbp)            # → (B,32)
        e_cal = torch.cat([e_sbp, e_dbp], dim=-1)   # → (B,64)

        # 4) fuse
        h = torch.cat([delta_f, e_cal], dim=-1)     # → (B,192)

        # 5) final MLP
        x = F.relu(self.bn_fuse1(self.fc_fuse1(h))) # → (B,128)
        out = self.fc_fuse2(x)                      # → (B,2) = (SBP_norm, DBP_norm)

        return out                                  # these are normalized outputs
```

You would wrap that in a **`Trainer`** which minimizes, say, mean-squared error:

```
python

loss = F.mse_loss(pred[:,0], sbp_true) + F.mse_loss(pred[:,1], dbp_true)
# or any weighted combination.
```

---

9. How to Store & Load Your Data “Per Patient”

1. **On Disk** (for patient “123”):

```
python

# patient_123.npz
np.savez_compressed(
    "patient_123.npz",
    cal_ppg = cal_ppg_array,       # shape (500,)
    cal_sbp = cal_sbp_scalar,      # float
    cal_dbp = cal_dbp_scalar,      # float
    tgt_ppg = tgt_ppg_array,       # shape (N, 500)
    tgt_sbp = tgt_sbp_array,       # shape (N,)
    tgt_dbp = tgt_dbp_array,       # shape (N,)
    fs      = 50.0
)
```

- **`cal_ppg_array`**: a single 10 s PPG strip (500 samples).
- **`cal_sbp_scalar`**, **`cal_dbp_scalar`**: the SBP/DBP accompanying that calibration window.
- **`tgt_ppg_array`**: a 2D array of shape **`(N, 500)`**, containing N different 10 s windows (all at 50 Hz) that you wish to use as training/validation/test samples for this patient.
- **`tgt_sbp_array`**, **`tgt_dbp_array`**: two length-N vectors giving the ground-truth SBP/DBP (in mmHg) for each corresponding row of **`tgt_ppg_array`**.
- **Note**: You *must* already have done all the filtering/QC (NaN-handling, SBP-range check, subject-balancing) before you build these arrays.

2. **During Training**: In your Dataset class (PyTorch) or data generator (TF), you do something like:

```
python

class PatientDataset(torch.utils.data.Dataset):
    def __init__(self, patient_npz_list, sbp_mu, sbp_sigma, dbp_mu, dbp_sigma):
        # patient_npz_list: list of strings like ["patient_123.npz", "patient_456.npz", ...]
        self.files = patient_npz_list
        self.sbp_mu, self.sbp_sigma = sbp_mu, sbp_sigma
        self.dbp_mu, self.dbp_sigma = dbp_mu, dbp_sigma

    def __len__(self):
        return len(self.files)

    def __getitem__(self, idx):
        data = np.load(self.files[idx])
        cal_ppg = torch.Tensor(data["cal_ppg"])[None, :]  # → shape (1,500)
        sbp_cal = ( data["cal_sbp"] - self.sbp_mu ) / self.sbp_sigma
        dbp_cal = ( data["cal_dbp"] - self.dbp_mu ) / self.dbp_sigma

        tgt_ppg_all = torch.Tensor(data["tgt_ppg"])       # → shape (N, 500)
        tgt_sbp_all = torch.Tensor(data["tgt_sbp"])       # → (N,)
        tgt_dbp_all = torch.Tensor(data["tgt_dbp"])       # → (N,)

        # 1) pick a random window i from [0..N-1]:
        i = np.random.randint(0, tgt_ppg_all.shape[0])
        tgt_ppg = tgt_ppg_all[i:i+1, :]                   # → shape (1,500)
        sbp_tgt = (tgt_sbp_all[i] - self.sbp_mu) / self.sbp_sigma
        dbp_tgt = (tgt_dbp_all[i] - self.dbp_mu) / self.dbp_sigma

        return {
            "target_ppg": tgt_ppg,    # shape (1,500)
            "cal_ppg":    cal_ppg,    # shape (1,500)
            "cal_sbp":    sbp_cal,    # scalar
            "cal_dbp":    dbp_cal,    # scalar
            "tgt_sbp":    sbp_tgt,    # scalar (label)
            "tgt_dbp":    dbp_tgt,    # scalar (label)
        }
```

Then in your training loop:

```
python

for batch in DataLoader(dataset, batch_size=B, shuffle=True):
    tgt_ppg = batch["target_ppg"].to(device)   # (B, 1, 500)
    cal_ppg = batch["cal_ppg"].to(device)      # (B, 1, 500)
    cal_sbp = batch["cal_sbp"].to(device)      # (B,)
    cal_dbp = batch["cal_dbp"].to(device)      # (B,)
    sbp_true= batch["tgt_sbp"].to(device)      # (B,)
    dbp_true= batch["tgt_dbp"].to(device)      # (B,)

    pred = model(tgt_ppg, cal_ppg, cal_sbp, cal_dbp)  # → (B, 2)
    sbp_pred, dbp_pred = pred[:,0], pred[:,1]         # each (B,)

    loss = F.mse_loss(sbp_pred, sbp_true) + F.mse_loss(dbp_pred, dbp_true)
    loss.backward()
```

```
        optimizer.step()
        optimizer.zero_grad()
```

That covers **exactly** what your data must look like on-disk and how it flows through the network at training time. By matching these shapes and blocks, you replicate the architecture in the paper faithfully.

Key Takeaways

1. **Per-patient `.npz` must contain**:
   - A single calibration PPG window + its SBP/DBP,
   - A variable number N of target PPG windows + their SBP/DBP.
2. **Three "sub-nets" share weights**:
   - The 1D-CNN is called on both target and calibration PPG (identical weights).
   - The MLP for SBP and DBP calibration *shares* weights between those two scalar branches.
3. **Fusion** is done by:
   - $|f_{\text{tgt}} - f_{\text{cal}}|$ (128-dim)
   - Concatenate with MLPcal output (64-dim)
   - Dense→128–>ReLU → Dense→2.
4. **Normalization** of SBP/DBP is crucial:
   - Compute μ, σ on training-set labels.
   - Convert all SBP/DBP to `(SBP–μ)/σ` before passing into the network.
   - Reverse at inference.
5. **Batch composition**:
   - Each batch has B **different** patients.
   - For each patient, you randomly pick exactly one of that patient's N target windows.
   - The "calibration PPG + (SBP_cal, DBP_cal)" is the same for all B "samples" from that patient within an epoch.

Once you follow this data-format and network blueprint exactly, you will have reproduced the PPG2BP-Net architecture (Fig. 4 of Joung et al.) precisely.

can i have my final data for 1 patient in this format ? here can take either the 1st or from 1 to 10 any strip as the calib strip randomly and them use the same cablib to train my model for that patient . does that work

You can absolutely turn your existing `.npz` (which already contains all of one patient's 10 s strips and their SBP/DBP labels) into exactly the "calibration + target" format that Joung et al. expect. In other words, from your file:

```
graphql

d19cebf2-d90b-4ac5-aade-13d4929df4cf.npz
├─ ppg_windows   shape = (300, 500)    # 300 windows of 10 s × 50 Hz
├─ sbp_values    shape = (300,)        # SBP (mmHg) for each of those 300 windows
├─ dbp_values    shape = (300,)        # DBP (mmHg) for each of those 300 windows
└─ fs            scalar = 50.0
```

you can simply pick (randomly) one of those 300 indices to be your **calibration strip**, and treat the other 299 as "target" strips. Then save out a new `.npz` that looks like this (per patient):

```
perl

patient_<ID>.npz
├─ cal_ppg     shape = (500,)     # the single 10 s PPG used for calibration
├─ cal_sbp     scalar (float32)   # the corresponding SBP (mmHg) for that calibration window
├─ cal_dbp     scalar (float32)   # the corresponding DBP (mmHg) for that calibration window
├─ tgt_ppg     shape = (299, 500)  # the 299 remaining "target" PPG strips
├─ tgt_sbp     shape = (299,)     # SBP (mmHg) for each of those 299 targets
├─ tgt_dbp     shape = (299,)     # DBP (mmHg) for each of those 299 targets
└─ fs          scalar = 50.0
```

Below is a short Python snippet showing exactly how to do that transformation. You can rerun it each time you want to generate the "calibration + targets" file. In this example, we:

1. **Load** the "300-strip" file.
2. **Choose one index** at random (e.g. `cal_idx = 7`).
3. **Extract** that row as `(cal_ppg, cal_sbp, cal_dbp)`.
4. **Remove** it from the 300 stacks and call the remaining 299 rows `(tgt_ppg, tgt_sbp, tgt_dbp)`.
5. **Save** the new `.npz` under the names Joung et al. expect.

```
python

import numpy as np

# ================================================================================
# STEP 1: Load the "300-strip" patient file
# ================================================================================
orig = np.load("/mnt/data/d19cebf2-d90b-4ac5-aade-13d4929df4cf.npz")
ppg_windows = orig["ppg_windows"]   # shape (300, 500)
sbp_values  = orig["sbp_values"]    # shape (300,)
dbp_values  = orig["dbp_values"]    # shape (300,)
fs          = float(orig["fs"])     # 50.0

# ================================================================================
```

```
# STEP 2: Choose one strip to be the “calibration” window
#          (you can pick an index at random, or always pick index=0, or any fixed choice)
# ================================================================================
# Example: choose one index at random
np.random.seed(2023)              # for reproducibility
cal_idx = np.random.randint(0, ppg_windows.shape[0])  # e.g. 7

# ================================================================================
# STEP 3: Extract that “calibration” row
# ================================================================================
cal_ppg = ppg_windows[cal_idx, :]     # shape (500,)
cal_sbp = sbp_values[cal_idx]         # scalar (float)
cal_dbp = dbp_values[cal_idx]         # scalar (float)

# ================================================================================
# STEP 4: Build the “target” arrays by removing the calibration index
# ================================================================================
# Create boolean mask of length 300 that is False only at cal_idx:
mask = np.ones(ppg_windows.shape[0], dtype=bool)
mask[cal_idx] = False

tgt_ppg = ppg_windows[mask, :]   # shape (299, 500)
tgt_sbp = sbp_values[mask]       # shape (299,)
tgt_dbp = dbp_values[mask]       # shape (299,)

# ================================================================================
# STEP 5: Save into the new “calibration + targets” format
#           (exactly what Joung et al. expect per patient)
# ================================================================================
out_file = "/mnt/data/patient_d19cebf2_calib.npz"
np.savez_compressed(
    out_file,
    cal_ppg = cal_ppg.astype(np.float32),   # (500,)
    cal_sbp = np.float32(cal_sbp),          # scalar float32
    cal_dbp = np.float32(cal_dbp),          # scalar float32

    tgt_ppg = tgt_ppg.astype(np.float32),   # (299, 500)
    tgt_sbp = tgt_sbp.astype(np.float32),   # (299,)
    tgt_dbp = tgt_dbp.astype(np.float32),   # (299,)

    fs = np.float64(fs)                     # 50.0
)
print(f"Finished writing new file: {out_file}")
```

After you run this snippet exactly once, you’ll have:

```
graphql


/mnt/data/patient_d19cebf2_calib.npz
├─ cal_ppg      (500,)
├─ cal_sbp      (scalar)
├─ cal_dbp      (scalar)
├─ tgt_ppg      (299, 500)
├─ tgt_sbp      (299,)
├─ tgt_dbp      (299,)
└─ fs           (50.0)
```

How to Use That New File in Training

1. **Loading**:

```
python


data = np.load("/mnt/data/patient_d19cebf2_calib.npz")
cal_ppg = data["cal_ppg"]    # shape (500,)
cal_sbp = data["cal_sbp"]    # scalar
cal_dbp = data["cal_dbp"]    # scalar
tgt_ppg = data["tgt_ppg"]    # shape (299, 500)
```

```
tgt_sbp = data["tgt_sbp"]    # shape (299,)
tgt_dbp = data["tgt_dbp"]    # shape (299,)
fs      = data["fs"]         # 50.0
```

2. **Normalization** (important!):
   - Before training across *all* patients, compute **`μ_SBP, σ_SBP`** (and **`μ_DBP, σ_DBP`**) from every patient's **`tgt_sbp`** and **`tgt_dbp`** in your training set.
   - At training time, convert each **`cal_sbp`** → **`(cal_sbp − μ_SBP)/σ_SBP`** and each **`cal_dbp`** → **`(cal_dbp − μ_DBP)/σ_DBP`**. Also convert each **`tgt_sbp[i]`** and **`tgt_dbp[i]`** in the same way.
   - The network's final two outputs are then on the "normalized" scale; after you get its **`(pred_sbp_norm, pred_dbp_norm)`**, you invert the scaling:

     ```
     python

     pred_sbp_mmHg = pred_sbp_norm * σ_SBP + μ_SBP
     pred_dbp_mmHg = pred_dbp_norm * σ_DBP + μ_DBP
     ```

   - This zero-mean/unit-variance normalization is exactly what Joung et al. do before passing SBP/DBP into their MLP, and it helps the network converge faster.
3. **Batch Sampling** (one way to write your Dataset class):

```
python

class OnePatientDataset(torch.utils.data.Dataset):
    def __init__(self, filepath, sbp_mu, sbp_sigma, dbp_mu, dbp_sigma):
        data = np.load(filepath)
        self.cal_ppg = torch.Tensor(data["cal_ppg"])[None, :]  # (1×500)
        self.cal_sbp = (data["cal_sbp"] - sbp_mu)/sbp_sigma    # scalar
        self.cal_dbp = (data["cal_dbp"] - dbp_mu)/dbp_sigma    # scalar

        self.tgt_ppg = torch.Tensor(data["tgt_ppg"])           # (299×500)
        sbp_raw = data["tgt_sbp"]                              # (299,)
        dbp_raw = data["tgt_dbp"]                              # (299,)
        self.tgt_sbp = torch.Tensor((sbp_raw - sbp_mu)/sbp_sigma)  # (299,)
        self.tgt_dbp = torch.Tensor((dbp_raw - dbp_mu)/dbp_sigma)  # (299,)

    def __len__(self):
        return self.tgt_ppg.shape[0]  # 299

    def __getitem__(self, i):
        # target PPG window #i, along with its SBP/DBP label:
        return {
            "target_ppg": self.tgt_ppg[i:i+1, :],  # shape (1,500)
            "cal_ppg":    self.cal_ppg,            # shape (1,500)
            "cal_sbp":    float(self.cal_sbp),     # scalar
            "cal_dbp":    float(self.cal_dbp),     # scalar
            "tgt_sbp":    float(self.tgt_sbp[i]),  # scalar (normalized)
            "tgt_dbp":    float(self.tgt_dbp[i]),  # scalar (normalized)
        }
```

Then in your training loop (PyTorch):

```
python

dataset = OnePatientDataset(
    "/mnt/data/patient_d19cebf2_calib.npz",
    sbp_mu, sbp_sigma, dbp_mu, dbp_sigma
)
loader = torch.utils.data.DataLoader(dataset, batch_size=B, shuffle=True)

for batch in loader:
```

```
    tgt_ppg = batch["target_ppg"].to(device)  # (B, 1, 500)
    cal_ppg = batch["cal_ppg"].to(device)     # (1, 500) → you may expand to (B, 1, 500) if need
    # If you want each sample in the batch to use the same cal_ppg, do:
    cal_ppg = cal_ppg.expand(B, -1, -1)       # now shape (B,1,500)

    cal_sbp = torch.tensor(batch["cal_sbp"], dtype=torch.float32).to(device)  # (B,)
    cal_dbp = torch.tensor(batch["cal_dbp"], dtype=torch.float32).to(device)  # (B,)

    sbp_true = torch.tensor(batch["tgt_sbp"], dtype=torch.float32).to(device)  # (B,)
    dbp_true = torch.tensor(batch["tgt_dbp"], dtype=torch.float32).to(device)  # (B,)

    # Forward pass through the network (see previous message for the model structure)
    pred = model(tgt_ppg, cal_ppg, cal_sbp, cal_dbp)  # → (B,2)

    pred_sbp = pred[:,0]   # (B,)
    pred_dbp = pred[:,1]   # (B,)

    loss = F.mse_loss(pred_sbp, sbp_true) + F.mse_loss(pred_dbp, dbp_true)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

A Few Additional Notes

- **Random versus Fixed Calibration**
  You asked: "Can I take either the 1st strip or picks strips 1–10 randomly as calibration, then use that same calibration to train on the rest?"
  - Yes. In the example above, we picked exactly one index (`cal_idx`) at random and treated it as calibration for all 299 "target" windows of that patient.
  - You could also pick a different calibration index each epoch (e.g. cycle through the first 10 windows), but Joung et al. typically use a **single** calibration window per patient.
  - If you use a "moving calibration" (randomly re-pick every epoch), make sure to also update `(cal_sbp, cal_dbp)` accordingly each time—otherwise the network will "see" multiple calibration labels for the same patient and might not stabilize.
- **Ensuring ≥ 50 Windows per Patient**
  Joung et al.'s paper discards any patient with fewer than 50 valid windows. Your file has 300 windows, so you're well over that. For other patients you load, if you find that `ppg_windows.shape[0] < 50`, you should drop them entirely from your training cohort.
- **Shuffling and Batching Across Patients**
  Ultimately, Joung et al. train on many patients at once. In practice, your PyTorch `DataLoader` or Keras `Sequence` will loop over a list of "per-patient" files. Each minibatch might contain one sample from each of, say, 32 different patients (so the loader needs to pick 32 files, load just one random "target" example from each, plus its shared calibration). This guarantees that each gradient update sees a diverse set of subjects. If you only have one patient so far, you can still train, but you will not generalize.
- **Conclusion**
  By reorganizing the single 300-strip file into "1 calibration strip + 299 target strips," storing it as

```
makefile

patient_<ID>.npz:
  cal_ppg, cal_sbp, cal_dbp,
```

```
    tgt_ppg, tgt_sbp, tgt_dbp,
    fs
```

you have exactly the format required by Joung et al.'s Figure 4. During training, you always feed the network:

1. One new 10 s PPG (**`target_ppg`**),
2. The same calibration PPG (**`cal_ppg`**) plus its two scalars (**`cal_sbp`**, **`cal_dbp`**),

and train to predict that new window's **`(SBP, DBP)`**.

That approach will work—just make sure you perform the same downstream label-normalization, subject-splitting, and per-patient-balancing (≥ 50, ≤ 100 windows) that the paper prescribes. Good luck!

or else instead of doing this can i just tell the model that while triang for all patinets use the 4th strip as the calib ?

Yes. Instead of randomly picking a calibration strip per patient, you can simply agree (once and for all) to use, say, the **4th 10 s window** of every patient as your calibration. As long as that 4th strip is (a) physiologically valid (no NaNs, flat-lines, extreme SBP/DBP) and (b) remains the same choice for that patient during all epochs, the network will learn exactly the same "calibration→target" mapping that Joung et al. describe. Below is a short outline (and code snippet) showing how you would do it:

---

1. Why a Fixed-Index Calibration Works

1. **Reproducibility:**
   Joung et al. picked one 10 s PPG segment per subject—once—treating it as the "known SBP/DBP" for that subject. They never changed it. By choosing a fixed index (e.g. always the 4th window), you do exactly the same thing.
2. **No Information Leakage:**
   As long as you do not use the 4th strip's SBP/DBP to train on any *other* patient, and you never switch calib-windows mid-training, there is no leakage. At inference time, you will feed in that same 4th strip (its normalized SBP/DBP) as the calibration input whenever you predict for that patient's new windows.
3. **Simplicity:**
   You don't have to randomly sample, you don't incur any overhead of re-loading multiple "possible" calib strips. The only requirement is that the 4th strip truly is "valid" (i.e. it passed all QC, has no NaNs, and its SBP/DBP is in your 70 – 180 mmHg band).

---

2. How to Reformat Your Data for "4th-Strip Calibration"

Assume your original per-patient `.npz` still looks like this:

```
graphql

d19cebf2-d90b-4ac5-aade-13d4929df4cf.npz
├─ ppg_windows   (300, 500)    # 10 s windows at 50 Hz
├─ sbp_values    (300,)        # SBP in mmHg for each of those 300 windows
├─ dbp_values    (300,)        # DBP in mmHg for each of those 300 windows
└─ fs            scalar (50.0)
```

You want to fix **index 3** (zero-based, so it's the "4th strip") as your calibration for that patient. Then "the other 299 strips" become your training targets. Here is a snippet that:

1. Loads the 300-strip file.
2. Takes row `idx=3` as `(cal_ppg, cal_sbp, cal_dbp)`.
3. Uses the other 299 rows as `(tgt_ppg, tgt_sbp, tgt_dbp)`.

4. Saves to **patient_<ID>_cal4.npz** in the exact format Joung et al. expect.

```
python

import numpy as np

# ──────────────────────────────────────────────────────────────────────────────
# 1) Load the original 300-strip file for this one patient
# ──────────────────────────────────────────────────────────────────────────────
orig = np.load("/mnt/data/d19cebf2-d90b-4ac5-aade-13d4929df4cf.npz")
ppg_windows = orig["ppg_windows"]   # shape (300, 500)
sbp_values  = orig["sbp_values"]    # shape (300,)
dbp_values  = orig["dbp_values"]    # shape (300,)
fs          = float(orig["fs"])     # should be 50.0

# ──────────────────────────────────────────────────────────────────────────────
# 2) Fix index=3 as the “4th strip” (zero-based) for calibration
# ──────────────────────────────────────────────────────────────────────────────
cal_idx = 3
cal_ppg = ppg_windows[cal_idx, :]   # shape (500,)
cal_sbp = sbp_values[cal_idx]       # scalar, in mmHg
cal_dbp = dbp_values[cal_idx]       # scalar, in mmHg

# ──────────────────────────────────────────────────────────────────────────────
# 3) Build the “target” arrays by removing that calibration index
# ──────────────────────────────────────────────────────────────────────────────
mask = np.ones(ppg_windows.shape[0], dtype=bool)
mask[cal_idx] = False

tgt_ppg = ppg_windows[mask, :]   # shape (299, 500)
tgt_sbp = sbp_values[mask]       # shape (299,)
tgt_dbp = dbp_values[mask]       # shape (299,)

# ──────────────────────────────────────────────────────────────────────────────
# 4) Save it in PPG2BP-Net’s expected “per-patient” format
# ──────────────────────────────────────────────────────────────────────────────
out_file = "/mnt/data/patient_d19cebf2_cal4.npz"
np.savez_compressed(
    out_file,
    cal_ppg = cal_ppg.astype(np.float32),   # (500,)
    cal_sbp = np.float32(cal_sbp),          # scalar
    cal_dbp = np.float32(cal_dbp),          # scalar

    tgt_ppg = tgt_ppg.astype(np.float32),   # (299, 500)
    tgt_sbp = tgt_sbp.astype(np.float32),   # (299,)
    tgt_dbp = tgt_dbp.astype(np.float32),   # (299,)

    fs = np.float64(fs)                     # 50.0
)
print(f"Saved “4th-strip” calibration file to: {out_file}")
```

Once you do this for every patient (loop over all your 300-strip **.npz** files, always using **idx=3**), you end up with a set of files:

```
python-repl

patient_<ID>_cal4.npz   # one per patient
...
```

Each contains exactly:

```
graphql

├─ cal_ppg   shape=(500,)
├─ cal_sbp   scalar
├─ cal_dbp   scalar
├─ tgt_ppg   shape=(299, 500)
```

```
├─ tgt_sbp   shape=(299,)
├─ tgt_dbp   shape=(299,)
└─ fs        scalar = 50.0
```

That is **exactly** the format Joung et al. used in their Figure 4, except they called it "calibration PPG + SBP/DBP" and "target PPG + SBP/DBP."

---

3. Training Loop with Fixed 4th-Strip Calibration

In your PyTorch (or TensorFlow) data loader, you now simply:

1. Load `patient_<ID>_cal4.npz`.
2. **Normalize**
   - Compute global training-set means (μ_SBP, μ_DBP) and standard deviations (σ_SBP, σ_DBP) once, across *all* patients' `tgt_sbp` & `tgt_dbp`.
   - Convert each patient's `cal_sbp → (cal_sbp − μ_SBP)/σ_SBP` and `cal_dbp → (cal_dbp − μ_DBP)/σ_DBP`.
   - Convert each row in `tgt_sbp, tgt_dbp` the same way.
3. Construct a Dataset whose `__getitem__(i)` returns for the *i-th target window* of that patient:

```python
{
  "target_ppg":  float32 array shape (1, 500),
  "cal_ppg":     float32 array shape (1, 500),  # the 4th-strip, same for all i
  "cal_sbp":     scalar float32 (normalized),
  "cal_dbp":     scalar float32 (normalized),
  "tgt_sbp":     scalar float32 (normalized target SBP),
  "tgt_dbp":     scalar float32 (normalized target DBP),
}
```

4. Each training iteration picks a batch of `B` different patients, each with one random "target" index **and** that patient's fixed calibration inputs.

Because you never change the calibration index, the model will learn exactly how "if PPG looks like cal_ppg and SBP_cal=..., DBP_cal=..., then for new PPG=target_ppg, predict SBP, DBP."

---

4. Pros and Cons of Always Using the 4th Strip

- **Pros**
  1. **Simplicity**: You only ever store one calibration window per patient.
  2. **Reproducibility**: Everyone knows "we used the 4th 10 s for calibration," so it's clear and unambiguous.
  3. **Matches the Paper**: Joung et al. used one fixed calibration segment (they often picked the first clean 10 s). Using the 4th is the same idea.
- **Cons**

1. **If the 4th strip is occasionally bad** (e.g. motion artifact, flatline), you'll have a "bad calibration," and every subsequent target window for that patient will use that defective calibration. You must ensure the 4th strip **always** passed your QC (no NaNs, good SBP/DBP range).
2. **Lack of variability**: Some patients' blood pressure drifts over the anesthetic. If you always calibrate with a 10 s snapshot taken early in the surgery, the calibration SBP/DBP may be far from the "true" SBP/DBP later. Joung et al. assume calibration happens at the start of each recording; if you want to handle "drifting" BP, you'd need multiple calibrations over time.
3. **Subject-specific scaling**: The network expects that "cal_ppg" + "(SBP_cal,DBP_cal)" come from the *same* time that you measure them. If the 4th strip is taken at minute 40 of surgery, but your target windows span minutes 20–200, only those target windows recorded *after* minute 40 will benefit from a calibration that actually matches them. Early windows (minute 25) might have different hemodynamics than minute 40. In practice Joung et al. train/test within a fairly narrow time range (all 10 s windows taken within a stable BP period).

If you are content with **one calibration per patient** and your entire set of target windows is recorded roughly at the same stable BP that you measured in the 4th strip, then "always use index 3" is perfectly fine.

---

5. Checklist Before You Train

1. **Verify QC of the 4th Strip**
   - Inspect window 3 (the 4th window) to ensure no NaNs, no flatline, SBP∈[70,180], DBP∈[40,120], etc. If it fails QC, pick a different fixed index (e.g. window 0 or 1).
2. **Normalize SBP/DBP Using Only Training Set Statistics**
   - After you build all `patient_*_cal4.npz` files for every patient, gather all their `tgt_sbp` values, compute μ_SBP, σ_SBP on **only** the training patients, and do the same for DBP.
   - Use these μ/σ to standardize both `cal_sbp` and each patient's entire `tgt_sbp` vector. Do not include validation/test in that calculation.
3. **Subject-Independent Splits**
   - Make sure your train/val/test split is by patient, not by window. If patient A is in the training set, none of A's windows (nor A's calibration) should appear in validation or test.
4. **Batch Composition**
   - If your DataLoader batches by sampling **one** window from each of `B` patients at a time, then your "calibration input" for each of those `B` examples is the same 4th strip for that patient. In code, you simply do:

```
python

cal_ppg = cal_ppg.expand(B, -1, -1)   # replicate the 4th-strip PPG across the batch
```

5. **Model Definition**
   - Ensure your CNN/Multi-block architecture matches Fig. 4 exactly: five 1D-conv layers, average-pool in the right place, flatten → Dense(128) → BN → ReLU → output embedding, then two-

branch MLP for SBP_cal & DBP_cal, then fusion, then final Dense→2.

Once you confirm that, you are ready to train. At each training step, you will:

- **Input**:

```
css

target_ppg [B×1×500],   cal_ppg [B×1×500],
cal_sbp [B],             cal_dbp [B]
```

  (All normalized except the raw PPGs.)

- **Output**:

```
css

pred_sbp_norm [B],  pred_dbp_norm [B]
```

- **Loss** (example):

```
python

loss = MSE(pred_sbp_norm, true_sbp_norm) + MSE(pred_dbp_norm, true_dbp_norm)
```

- **Inference**:
  After training, to recover actual mmHg, do:

```
python

sbp_pred_mmHg = pred_sbp_norm * σ_SBP + μ_SBP
dbp_pred_mmHg = pred_dbp_norm * σ_DBP + μ_DBP
```

In Summary

- **Yes**, you can fix the 4th strip as your calibration for each patient.
- Just be sure that strip is good (passes all QC).
- Re-save each patient's 300 windows into a new file of the form:

```
text

patient_XXX_cal4.npz
├ cal_ppg   (500,)
├ cal_sbp   scalar
├ cal_dbp   scalar
├ tgt_ppg   (299, 500)
├ tgt_sbp   (299,)
├ tgt_dbp   (299,)
└ fs        50.0
```

- During training, always feed the network `(target_ppg, cal_ppg, cal_sbp_norm, cal_dbp_norm)` and regress `(true_sbp_norm, true_dbp_norm)`.

- Provided you also perform SBP/DBP normalization, subject-independent splitting, and balancing (≥50 windows per patient), this will exactly replicate Joung et al.'s calibration strategy—just with a "fixed-index" version instead of "pick the first valid strip." Good luck!

```
# downsample the result from 500hz to 50hz and save as a new .npz file
#!/usr/bin/env python3
import argparse
import numpy as np

def downsample_to_50hz(input_path: str, output_path: str) -> None:
    """
    Load 'ppg', 'abp', and 'fs' from input_path (.npz),
    downsample both signals from 500 Hz to 50 Hz by taking every 10th sample,
    and save the downsampled arrays (plus 'fs'=50) into output_path (.npz).
    """
    data = np.load(input_path)
    ppg = data["ppg"]       # shape: (L,)
    abp = data["abp"]       # shape: (L,)
    fs = float(data["fs"])  # sampling frequency, expected 500.0

    if ppg.shape != abp.shape:
        raise ValueError("'ppg' and 'abp' must have the same shape.")

    if int(fs) != 500:
        raise ValueError(
            f"Expected original fs=500 Hz, but got fs={fs}."
        )

    # Compute downsampling factor
    target_fs = 50
    factor = int(fs // target_fs)
    if fs / factor != target_fs:
        raise ValueError(
            f"Downsampling factor must be an integer. Got fs={fs}, "
            f"target_fs={target_fs}, fs/factor={fs/factor}."
        )

    # Use slicing to take every 'factor'-th sample (view, memory-efficient)
    ppg_ds = ppg[::factor]
    abp_ds = abp[::factor]

    # Release memory of the originals
    del ppg, abp, data

    # Save the downsampled signals with new fs
    np.savez_compressed(
        output_path,
        ppg=ppg_ds,
        abp=abp_ds,
        fs=float(target_fs)
    )
```

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Downsample PPG/ABP from 500 Hz to 50 Hz in a .npz file."
    )
    parser.add_argument(
        "input_npz",
        help="Path to the cleaned .npz (containing 'ppg', 'abp', and 'fs=500')."
    )
    parser.add_argument(
        "output_npz",
        help="Path where the downsampled .npz (fs=50) will be saved."
    )
    args = parser.parse_args()
    downsample_to_50hz(args.input_npz, args.output_npz)
    print(f"Saved downsampled signals to '{args.output_npz}'.")


#chmod +x downsample_to_50hz.py
#./downsample_to_50hz.py cleaned_file.npz downsampled_file.npz

is this corre t ?
```

Your script will mechanically take every 10th sample and save a new 50 Hz file, but as written it will likely introduce aliasing artifacts. In other words, simply doing:

```
python

ppg_ds = ppg[::10]
abp_ds = abp[::10]
```

will work “in the sense” of reducing the sample rate from 500 Hz → 50 Hz, but any frequency content above 25 Hz (for example, motion spikes or mains noise harmonics) will fold back into the 0–25 Hz band. In physiologic PPG/ABP data, most of the useful energy is indeed below ~15 Hz, but that doesn’t guarantee there isn’t some high-frequency artifact that you want to remove first. The paper you’re following (Joung et al.) explicitly smooths or low-passes before decimation, which prevents aliasing.

---

1. What your code does correctly

1. **Checks that `fs == 500`**

   ```
   python

   if int(fs) != 500:
       raise ValueError(f"Expected original fs=500 Hz, but got fs={fs}.")
   ```

   This is good—if your input isn’t actually 500 Hz, you catch it immediately.

2. **Computes an integer downsample factor**

```
python

target_fs = 50
factor = int(fs // target_fs)  # factor = 500 // 50 = 10
if fs / factor != target_fs:
    raise ValueError("Downsampling factor must be an integer…")
```

So you ensure that `500 / 10 == 50`, then slice by `ppg[::10]`.

3. **Saves out a new `.npz` with `fs = 50.0`**

```
python

np.savez_compressed(
    output_path,
    ppg = ppg_ds,
    abp = abp_ds,
    fs  = float(target_fs)
)
```

That matches the expected format for all downstream scripts (they look for `"ppg"`, `"abp"`, and `"fs"`).

---

2. Why you should add an anti-alias (low-pass) filter first

Whenever you downsample (especially by a factor of 10), any frequency content above the new Nyquist (25 Hz) will "fold" back below 25 Hz. In practice:

- **ABP** can contain small high-frequency ripples (e.g. line noise at 50 Hz, or electronic noise) that you don't want aliasing into your 0–25 Hz band.
- **PPG** often has very little power above ~15 Hz, but motion-artifacts or sensor noise can introduce higher frequencies. Folding them back in can distort your pulse shape.

How to fix this

Right before you slice, insert a zero-phase low-pass filter (e.g. a 4th-order Butterworth at 15 Hz or Savitzky–Golay smoothing) to attenuate > 25 Hz. For example:

```
python

from scipy.signal import butter, filtfilt

def lowpass_filter(x: np.ndarray, cutoff_hz: float, fs: float, order: int = 4) -> np.ndarray:
    nyq = fs / 2
    b, a = butter(order, cutoff_hz / nyq, btype="lowpass")
    return filtfilt(b, a, x)

def downsample_to_50hz(input_path: str, output_path: str) -> None:
    data = np.load(input_path)
    ppg = data["ppg"]
    abp = data["abp"]
    fs  = float(data["fs"])

    if int(fs) != 500:
        raise ValueError(f"Expected original fs=500 Hz, but got fs={fs}.")

    # 1) Low-pass at, say, 15 Hz to remove everything above 25 Hz Nyquist
    ppg_filt = lowpass_filter(ppg, cutoff_hz=15.0, fs=fs, order=4)
    abp_filt = lowpass_filter(abp, cutoff_hz=15.0, fs=fs, order=4)
```

```
    # 2) Downsample by factor of 10
    ppg_ds = ppg_filt[::10]
    abp_ds = abp_filt[::10]

    np.savez_compressed(
        output_path,
        ppg = ppg_ds,
        abp = abp_ds,
        fs  = float(50.0)
    )
    print(f"Saved downsampled signals to '{output_path}'.")
```

- **`butter(order, cutoff_hz/nyq, btype="lowpass")`** designs a digital Butterworth filter with a cutoff at 15 Hz (well below the new Nyquist of 25 Hz).
- **`filtfilt(b, a, x)`** applies it forward and backward, giving zero-phase distortion (so your PPG/ABP upstrokes don't shift in time).
- Then you safely do **`x[::10]`** without aliasing.

---

3. Summary of corrections

1. **Original code (aliasing risk):**

```
python

ppg_ds = ppg[::10]
abp_ds = abp[::10]
```

– This will drop high-frequency data into the 0–25 Hz band.

2. **Recommended fix (low-pass → decimate):**

```
python

# 1) Apply lowpass filter at cutoff < 25 Hz (e.g. 15 Hz).
ppg_filt = lowpass_filter(ppg, cutoff_hz=15.0, fs=500.0)
abp_filt = lowpass_filter(abp, cutoff_hz=15.0, fs=500.0)

# 2) Then take every 10th sample:
ppg_ds = ppg_filt[::10]
abp_ds = abp_filt[::10]
```

With that addition, your script will be both "formally correct" (it still only downsamples by slicing) and also "practically correct" (it won't alias unwanted high-frequency components into your 50 Hz signal).

Final, Revised Script

```
python

#!/usr/bin/env python3
import argparse
import numpy as np
from scipy.signal import butter, filtfilt

def lowpass_filter(x: np.ndarray, cutoff_hz: float, fs: float, order: int = 4) -> np.ndarray:
    """
    Zero-phase low-pass filter: attenuate all frequencies above 'cutoff_hz'.
    """
```

```
    nyq = fs / 2
    b, a = butter(order, cutoff_hz / nyq, btype="lowpass")
    return filtfilt(b, a, x)

def downsample_to_50hz(input_path: str, output_path: str) -> None:
    """
    1) Load 'ppg', 'abp', 'fs' from input_path (.npz),
    2) Low-pass both at cutoff ~15 Hz to remove >25 Hz content,
    3) Decimate by taking every 10th sample (500 -> 50 Hz),
    4) Save as new .npz with fs=50.0.
    """
    data = np.load(input_path)
    ppg = data["ppg"]
    abp = data["abp"]
    fs  = float(data["fs"])

    if ppg.shape != abp.shape:
        raise ValueError("'ppg' and 'abp' must have the same shape.")
    if int(fs) != 500:
        raise ValueError(f"Expected original fs=500 Hz, but got fs={fs}.")

    # 1) Anti-alias via low-pass (zero-phase) at 15 Hz
    ppg_filt = lowpass_filter(ppg, cutoff_hz=15.0, fs=fs, order=4)
    abp_filt = lowpass_filter(abp, cutoff_hz=15.0, fs=fs, order=4)

    # 2) Downsample by integer factor 10
    ppg_ds = ppg_filt[::10]
    abp_ds = abp_filt[::10]

    # 3) Save new 50 Hz file
    np.savez_compressed(
        output_path,
        ppg = ppg_ds,
        abp = abp_ds,
        fs  = float(50.0)
    )
    print(f"Saved downsampled signals to '{output_path}'.")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Downsample PPG/ABP from 500 Hz to 50 Hz (with low-pass)."
    )
    parser.add_argument(
        "input_npz",
        help="Path to the cleaned .npz (must contain 'ppg','abp','fs=500')."
    )
    parser.add_argument(
        "output_npz",
        help="Where to save the downsampled .npz (ppg,abp at 50 Hz + 'fs')."
    )
    args = parser.parse_args()
    downsample_to_50hz(args.input_npz, args.output_npz)
```

With this revision you'll have a truly correct downsampling step—identical in spirit to what Joung et al. do when they "Savitzky–Golay" or low-pass filter before decimating from 500 → 50 Hz.

give me full code

```python
#!/usr/bin/env python3
import argparse
import numpy as np
from scipy.signal import butter, filtfilt

def lowpass_filter(x: np.ndarray, cutoff_hz: float, fs: float, order: int = 4) -> np.ndarray:
    """
    Zero-phase low-pass filter: attenuate all frequencies above 'cutoff_hz'.
    """
    nyq = fs / 2
    b, a = butter(order, cutoff_hz / nyq, btype="lowpass")
    return filtfilt(b, a, x)

def downsample_to_50hz(input_path: str, output_path: str) -> None:
    """
    1) Load 'ppg', 'abp', 'fs' from input_path (.npz),
    2) Low-pass both at cutoff ~15 Hz to remove >25 Hz content,
    3) Decimate by taking every 10th sample (500 → 50 Hz),
    4) Save as new .npz with fs=50.0
```