Audio → MIDI: Klassisch & KI (Flöte, Viertelnoten, Oktave)

Dieses Dokument beschreibt zwei Wege, um aus monophonem Audio (z. B. Flöte) MIDI zu erzeugen:  
1) Klassische Signalverarbeitung mit pYIN, Quantisierung und Monophonie (a2m.py).  
2) Ein kleines KI-Modell, das pro Viertel eine Tonklasse erkennt und direkt im Takt-Raster ausgibt (a2m-ai.py).  
Beide Ansätze sind praxiserprobt und in deinem Repo enthalten.

# 1) Klassischer Weg – a2m.py

Echte Grundfrequenz (f₀) aus dem Audio schätzen (pYIN), daraus Noten formen, am musikalischen Raster (z. B. Viertel) ausrichten und als MIDI schreiben.

## Pipeline

* Audio laden → Mono, gewünschte Abtastrate (z. B. 22050 Hz).
* Pitch-Tracking (librosa.pyin) → pro Frame Grundfrequenz f₀ (Hz) oder unvoiced (NaN).
* Rohnoten bilden → zusammenhängende Frames ähnlicher Tonhöhe werden zu Noten (Pitch, Start, Ende, Velocity aus RMS).
* Bar-Blips entfernen → sehr kurze Überhänge/Legato-Reste direkt an der Taktgrenze entfernen (verhindert Doppelnoten).
* Globales Alignment + Quantisierung → Start/Ende ans musikalische Raster (z. B. Viertel) schieben; globalen Versatz aus Onsets schätzen.
* Monophonie erzwingen → Überlappungen entfernen, direkt angrenzende gleiche Pitches zusammenkleben.
* MIDI schreiben → GM-Instrument (z. B. 73 = Flöte), optional Transponieren (+/- Halbtöne).

## Beispiel (Kommandozeile)

python a2m.py floete.wav floete.mid \  
 --fmin C4 --fmax C7 --instrument 73 --transpose 12 \  
 --bpm 130 --quant 1 --sig\_num 4 --sig\_den 4 \  
 --bar\_blip\_ms 220 --bar\_eps\_ms 35 --overlap\_eps\_ms 2 --gap\_ms 0

## Wichtige Parameter

* Tonhöhenbereich: `--fmin/--fmax` (z. B. C4..C7) – eng wählen für stabilere pYIN-Ergebnisse.
* Zeitauflösung: `--sr 22050 --hop 256 --win 2048` (~11.6 ms/Frame).
* Blip-Filter an der Barline: `--bar\_blip\_ms 200–240`, `--bar\_eps\_ms 30–40`.
* Quantisierung: `--bpm`, `--quant 1` (Viertel), `--quant\_mode nearest|floorceil`.
* Alignment: grundsätzlich aktiv lassen; `--no\_align` nur zum Testen.
* Monophonie-Toleranz: `--overlap\_eps\_ms 2–5`.
* NaN-Lücken: `--gap\_ms` (bei Legato oft 0).

## Stärken

* Erkennt echte Tonhöhen inkl. Vibrato/Glissando.
* Transparent und gut über Parameter kontrollierbar.

## Grenzen

* Empfindlicher gegenüber Rauschen/Legato – benötigt oft Blip-Filter & Quantisierung.
* Timing hängt von f₀-Stabilität ab.

## Troubleshooting

* Noten starten „zwischen“ Vierteln → `--bpm` prüfen; Auto-Alignment aktiv lassen.
* Doppelte Note am Taktanfang → `--bar\_blip\_ms` erhöhen; `--bar\_eps\_ms` auf ~35 ms stellen.
* Zu viele Mini-Noten → `--min\_frames` erhöhen; `--fmin/--fmax` enger; ggf. `--hop` etwas größer.

# 2) KI-Weg – a2m-ai.py

Audio wird strikt im Viertel-Grid (Start = 0.0 s) geschnitten. Ein kleines CNN klassifiziert pro Viertel eine von 13 Klassen: 12 Töne einer Oktave (base\_midi..base\_midi+11) + REST.

## Bestandteile

* mkdata: erzeugt synthetische Trainingsdaten (kleiner Flöten-Synth, Augmentations: Gain, leichter Tilt‑EQ, Rauschen, ±3 % Time‑Stretch).
* train: trainiert ein kleines CNN (3×Conv, GAP, Linear) mit Live-Status und Epochen‑Checkpoints (`…epXX.pt`); bestes Modell `model.pt`.
* infer: exaktes Grid (Start=0), volle Takte (per `--bars` oder Auto-Rundung), Ende wird bei Bedarf mit Stille gepolstert.

## Beispiele (Kommandozeile)

python a2m-ai.py mkdata --out data\_quarters.npz --bpm 130 --base\_midi 60

set PYTHONUNBUFFERED=1 && python -u a2m-ai.py train --data data\_quarters.npz --out model.pt --epochs 20

python a2m-ai.py infer \  
 --wav floete.wav --out floete.mid --model model.pt \  
 --bpm 130 --base\_midi 60 --bars 4

## Optionen

* mkdata: `--n\_sequences`, `--bars`, `--bpm`, `--base\_midi`, `--rest\_prob`, `--sr`, `--seed`.
* train: `--epochs`, `--batch`, `--lr`, `--val\_split`, `--max\_samples`, `--resume`, `--patience`.
* infer: `--bpm`, `--base\_midi`, `--sr`, `--bars` (erzwingt exakte Taktzahl).

## Trainingsdaten (NPZ-Format)

* X: (N, 1, 64, 96) float32 — Log‑Mel‑Patches pro Viertel
* y: (N,) int64 — Labels: 0..11 = Ton (base\_midi+offset), 12 = REST
* meta: { "bpm": ..., "base\_midi": ..., "sr": ... }

## Stärken

* Perfektes Timing (Start=0, volles Viertel‑Raster).
* Robust gegen Legato/Overlaps; sehr schnelle Inferenz (auch CPU).

## Grenzen

* Notenlänge ist Viertel‑rasterbasiert (lange Noten entstehen als mehrere gleiche Viertel).
* Kein Pitch‑Bend/Vibrato wie beim pYIN‑Weg.

## Laufzeit / Hardware

* Training: kleines Default‑Set → mehrere Stunden auf CPU; mit NVIDIA‑GPU deutlich schneller.
* Inferenz: Sekundenbereich.

# Glossar (ausführlich)

### CNN (Convolutional Neural Network)

Ein neuronales Netz, das mit kleinen Filtern über ein „Bild“ fährt und Muster erkennt. Unser „Bild“ ist ein Zeit‑Frequenz‑Bild des Tons (Log‑Mel‑Patch).

### Log‑Mel‑Patch

Ein kleiner Ausschnitt eines Spektrogramms pro Viertel (z. B. 64 Frequenzbänder × 96 Zeit‑„Pixel“). „Mel“ nähert die Wahrnehmung des Gehörs an; „Log“ bedeutet dB‑Skalierung.

### Augmentations

Zufällige Mini‑Veränderungen beim Training (z. B. Lautstärke, leichtes Rauschen, minimaler Time‑Stretch), damit das Modell robust wird und nicht nur einen einzigen Synth‑Sound „kennt“.

### Blip

Ein sehr kurzer, ungewollter Ton am Viertelanfang/-ende (z. B. durch Legato/Overlap). Im KI‑Ansatz stammen die Labels aus der Mitte des Viertels – so werden Blips am Rand ignoriert und doppelte Noten vermieden.

## Aufgabe / Labels

* Eine Oktave: 12 Töne + REST (= Stille) → 13 Klassen.
* Jede Audio‑Scheibe ist genau ein Viertel lang (60/BPM Sekunden). Für jede dieser Scheiben berechnen wir ein Log‑Mel‑Patch und lassen das CNN eine Klasse vorhersagen.
* Label‑Regel (KI): Die Note, die die Mitte des Viertels überdeckt, ist das Ziel‑Label. So werden Blips/Legato am Rand nicht „mit‑gelabelt“.

# Installation (kurz)

* requirements.txt: numpy, scipy, librosa, soundfile, pretty\_midi, tqdm, torch

CPU: pip install --index-url https://download.pytorch.org/whl/cpu -r requirements.txt

GPU: pip install --index-url https://download.pytorch.org/whl/cu129 -r requirements.txt (oder cu128/nightly, je nach Verfügbarkeit)

* Linux: ggf. sudo apt-get install -y libsndfile1

# Wann welchen Weg?

|  |  |
| --- | --- |
| Situation | Empfehlung |
| Exakte Tonhöhen (auch außerhalb der Oktave), Vibrato/Glissando | Klassisch (a2m.py) |
| Ultra‑stabiles Viertel‑Timing, Oktave reicht | KI (a2m-ai.py) |
| Prototyp/Skizze, später in der DAW längere Noten „kleben“ | KI (a2m-ai.py) |
| Parametrische Kontrolle/Feintuning | Klassisch (a2m.py) |

# Nützliche .gitignore

\_\_pycache\_\_/  
\*.py[cod]  
\*.egg-info/  
.venv/  
venv/  
.idea/  
.vscode/  
.DS\_Store  
Thumbs.db  
\*.wav  
\*.flac  
\*.mp3  
\*.npz  
\*.pt  
checkpoints/

Erstellt am 2025-08-22 22:04:18