ChatGPT

Deep Learning Pipeline for 30-Second Music Genre Classification

Goal: Classify 30-second audio clips into 8 genres (with ~1000 clips per genre, 8000 total) using deep learning. We compare two approaches: (1) fine-tuning a **pretrained** audio model (e.g. AST or PANNs) and (2) training a **custom CNN from scratch** (optimized for CPU on a MacBook Pro 14"). We target the highest possible accuracy with an 80/20 train/validation split. Below we outline best practices for preprocessing, the training pipeline (data segmentation, augmentation, training schedule), a comparison of the approaches (accuracy and trade-offs), and how to deploy the trained model in a real application.

Preprocessing 30-Second Audio Clips

Proper preprocessing ensures the audio data is in a consistent, learnable format:

- **Sample Rate & Channels:** Convert audio to a single channel (mono) and a standard sample rate (44.1 kHz as specified). If files aren't already 44.1 kHz, resample them to 44100 Hz for consistency [1]. Using a consistent sampling rate also ensures compatibility with pretrained models' expectations [1]. Mono conversion simplifies analysis since stereo separation is usually not crucial for genre classification.

- **Trim/Padding:** Ensure each clip is 30 seconds. If some clips are longer, truncate them to 30s; if shorter, they can be looped or zero-padded (though in a controlled dataset, all clips are likely equal length).

- **Mel-Spectrogram Extraction:** Transform each audio clip into a log-mel spectrogram, which will serve as the input "image" for the neural network. This involves:

- Framing the audio with Short-Time Fourier Transform (STFT) windows. Use the given parameters: **FFT size 2048**, **hop length 512** (≈11.6 ms hop at 44.1 kHz), and **128 mel frequency bins**. This yields a time-frequency representation ~128×2580 in size for a 30s clip (since 30s / 11.6ms ≈ 2580 frames).
- **Mel scaling:** Map the power spectrum to the mel scale (128 mel bands) to emphasize perceptually relevant frequency spacing [2] [3].
- **Log amplitude:** Convert mel spectrograms to a log scale (e.g. decibels). Taking log (or `librosa.power_to_db`) compresses dynamic range so that both loud and soft parts are represented more evenly [4]. The result is a **log-mel spectrogram**, a 2D image capturing how frequency content evolves over 30 seconds.

- **Normalization:** Normalize or scale the spectrogram values for stable network training. A common approach is **scaling spectrogram pixel values to [0, 1] range** (after log-scaling) [5]. This can be done by subtracting the minimum and dividing by the range or by applying per-image normalization. Alternatively, one can standardize features (zero-mean, unit-variance) across the

dataset [6] [7], though simple min-max scaling is often sufficient for mel spectra. Perform this normalization **after** splitting into train/val to avoid data leakage [8].

- **Prepare inputs for CNN:** Treat each log-mel spectrogram as a single-channel image of shape (128, ~2580, 1). Add the channel dimension (1) to the array [9] and store the label. Also one-hot encode genre labels (or use integer labels if your framework handles them).

By following these steps, we convert each 30s audio clip into a standardized image-like input that a CNN or transformer can ingest [10] [5].

Training Pipeline Details

With preprocessed spectrograms, we set up the training pipeline. Key considerations include how to handle the long 30s sequence, data augmentation to expand effective data, and training scheduling (especially important on CPU):

- **Data Segmentation Strategy:** Thirty seconds of audio might be more information than a model needs at once, and feeding the full 128×2580 spectrogram can be memory-intensive. A best practice is to **split each 30s clip into shorter segments** for training:
- For example, cut each spectrogram into **5-second segments** (yielding 6 segments per clip, each ~128×430). This increases the training sample count (from 6400 training clips to ~38,400 segments if using 5s segments for all training clips) and helps the model learn from varied parts of the song. Many studies use this approach: e.g., one approach extracted **6 slices of 5s** from each 30s track to use as separate training samples [11]. Another approach used even more slices per track (e.g. 18 short excerpts) and combined their predictions [11].
- **Random Cropping:** Instead of fixed segmentation, you can randomly crop a 3–5s window from the 30s clip on each training epoch. This acts as augmentation – the model sees different parts of the song each epoch – and over many epochs it effectively learns from the full 30s without processing it all at once. This random-crop strategy helps expose the model to various sections (intro, chorus, etc.) and can improve generalization.
- **Track-level Prediction:** During validation/testing, to classify the full 30s clip, aggregate model predictions from multiple segments. For instance, take majority vote or average the predicted genre probabilities across 5–10 random segments spanning the clip. This ensemble-of-segments approach tends to boost accuracy by smoothing out any localized mispredictions [12] [13]. In fact, using an ensemble of slice predictions yielded **~92.7% accuracy** on a genre classification task by averaging outputs across a song [14].

- **Direct Full-Length Training:** It's also possible to train on the full 30s spectrogram by using a model architecture that can handle long inputs (e.g. CNN with global pooling). If using the full spectrogram, include sufficient pooling layers or a global average pooling so that the time dimension is reduced before the final dense layer. Given the added computational load of 30s inputs, the segmented approach above is often more efficient and equally effective (since short excerpts often contain enough genre cues [15]).

- **Data Augmentation:** To maximize effective data size and reduce overfitting (especially with only 8000 tracks), apply **audio augmentations** during training. Augmentation can significantly improve a from-scratch model's performance on limited data by creating new variations of the existing clips:

- **Waveform-level augmentations:** These include adding background noise or Gaussian noise, random gain adjustments (volume up/down), dynamic range compression or clipping, time stretching or compressing (altering tempo slightly without changing pitch), and pitch shifting (changing key within a small range). For example, one can compose multiple transforms such as adding Gaussian noise (signal-to-noise ratio in 10–20 dB), random gain ±6 dB, slight time stretch 0.8–1.2× speed, and pitch shifts up to ±2 semitones [16]. These preserve the genre characteristics while forcing the model to be robust to slight variations in speed, tuning, and noise.
- **Spectrogram-level augmentations:** Apply techniques like **SpecAugment**, e.g. randomly masking out frequency bands or time frames on the mel spectrogram. This helps the model rely on broad patterns rather than exact frequencies. One can zero out a couple of mel frequency bands and small time blocks in each spectrogram during training.
- **Mixup:** An advanced augmentation is **mixup**, where two spectrograms are mixed together with a random ratio and the model is trained on the mixed input with a correspondingly mixed label. This was used in fine-tuning PANNs and proved beneficial [17]. Mixup smooths decision boundaries and can improve generalization, though it makes genre labels less clear (it effectively trains the model to handle songs that could have characteristics of two genres).
- **Shift in time:** A subtle augmentation is shifting the audio slightly forward/backward in time (which for spectrogram means rolling the image in time dimension and maybe filling edges with silence). This can help the model not be overly sensitive to absolute position of a pattern in the 30s window.

Augmentations should be applied to training data only. Many libraries (like `audiomentations` or TorchAudio) can perform these on the fly [16]. With aggressive augmentation, the model can effectively see tens of thousands of unique training examples, mitigating overfitting.

- **Model Architecture (Custom CNN):** For the from-scratch approach on CPU, design a CNN that balances accuracy and efficiency:
- A feasible architecture is a **small 2D CNN** with 3–5 convolutional layers, each followed by pooling. For example, one might start with a layer of 16 or 32 filters (size 3×3) extracting low-level time-frequency features, then increase to 64, 128, etc. as depth increases [18]. Max-pooling layers can progressively reduce the time-frequency resolution (e.g. pool along time axis to handle the long 30s span). One architecture that worked on 30s mel-spectrograms used two conv layers (3×3 kernels) with 16 and 32 filters, each followed by pooling, then a dense layer of 64 units [18]. Our custom model can be a bit deeper: e.g. **4 conv layers** (32→64→128→128 filters) with 2×2 or 2×4 pooling after each, then a global average pooling and a dense output layer (8 neurons for 8 genres). We also include **batch normalization** (to stabilize training) and **ReLU** activations after conv layers for nonlinearity [19]. A **dropout** layer (e.g. 0.5 drop rate) before the final dense layer is recommended to reduce overfitting [19].
- Use **Adam optimizer** with a learning rate around 1e-3 (a typical starting point for scratch training) [20]. Monitor validation loss for early stopping. On CPU, a batch size of 8 or 16 might be necessary to fit memory, but ensure to shuffle batches.
- Training on CPU will be slower, but the small model and data augmentations help. Expect to train for on the order of **20–50 epochs**. Prior experiments on smaller genre datasets saw CNNs start overfitting after ~15 epochs without augmentation [21], but with augmentation and a larger dataset (8000 clips), you can train longer. Keep an eye on validation loss and stop when it begins to rise consistently (early stopping) to avoid overfitting.

- **Validation strategy:** Use the 20% validation split for monitoring accuracy after each epoch. Because of class balance (1000 each), a simple accuracy metric is fine, but also consider per-genre performance if needed to ensure no genre is left behind.

- **Model Setup (Pretrained Fine-Tuning):** For the pretrained approach, we leverage a model like **PANNs (CNN14)** or **AST** that comes pre-trained on a large audio dataset (like AudioSet). Fine-tuning involves:

- **Initialize** the model with pretrained weights. Replace the final output layer with a new dense layer of size 8 (for 8 genres) – this new layer will learn the genre classification.
- **Feature Extraction vs Full Fine-Tuning:** Since training on CPU is restrictive, one strategy is to **freeze** the majority of the pretrained model's layers initially (keep them fixed) and only train the new classification layer (and perhaps a couple of top layers) for a few epochs. This adapts the high-level features to the genre task quickly. Then, unfreeze some or all of the layers and continue training at a lower learning rate (e.g. 1e-4 or 5e-5) to fine-tune the feature representations to this specific music data [17]. This two-stage approach (freeze-then-unfreeze) can stabilize training and reduces initial computation.
- **Batch size** can be smaller for large models due to memory. On a MacBook CPU, you might use batch size 4–8. Alternatively, you could do feature extraction offline: pass all spectrograms through the pretrained model up to a certain layer to get "embeddings" and then train a small classifier on those – this is computationally lighter during training.
- **Training duration:** Pretrained models typically converge faster in terms of epochs. You might reach high accuracy in <10 epochs of fine-tuning since the model already learned relevant audio features [22] [23]. For example, fine-tuning PANNs on a genre dataset converged in around 2000 iterations (with batch 32) – roughly 50–60 epochs – to a high accuracy [24]. Monitor validation accuracy; with a low LR, you can train until performance plateaus. (Be mindful of not overfitting a huge model to 6400 training clips – heavy augmentation and possibly regularization like weight decay can help.)
- **Augmentation:** Fine-tuning can also benefit from the same augmentations described, though if the pretrained model was trained on diverse audio, it's already somewhat invariant. Still, adding noise or slight speed/pitch changes can help it adapt to the new dataset.

By the end of training, you'll have **two models**: (a) a smaller CNN trained from scratch, and (b) a fine-tuned large model. Next, we compare their performance and trade-offs.

Pretrained vs. Scratch Model: Accuracy & Trade-Offs

Accuracy Expectations: Fine-tuning a pretrained model will generally yield higher accuracy on this task than training a small CNN from scratch:

- Pretrained audio models (like AST or PANNs) have learned rich audio features from massive datasets, which transfers well to music genre recognition. For instance, fine-tuning a PANNs CNN14 model (pretrained on AudioSet) for music genre classification (GTZAN 10-genre dataset) achieved about **89% validation accuracy** [24]. With 8 genres and a larger dataset (FMA-small, 8000 clips), we can expect a similar or higher performance, potentially in the **85–95% accuracy** range with proper fine-tuning. In fact, state-of-the-art research models (e.g. hybrid CNN-Transformer architectures) have exceeded **90% accuracy on FMA-small** [25]. This suggests that a well-fine-tuned AST or PANNs could

likely reach *>90%* given enough training and augmentation, because the model can leverage high-level musical features it already knows (rhythmic patterns, instrument timbres, etc.) [22] [23] .

- A custom CNN trained from scratch will start with no prior knowledge and is limited by the dataset size. With 6400 training examples, a simple CNN might achieve moderate accuracy. Past attempts on the FMA-small dataset with basic CNNs have reported on the order of **50–60% accuracy** when no extensive augmentation or tuning was used [26] . For example, a baseline CNN achieved ~54% on 8-genre classification for FMA-small [26] , and even adding an RNN only brought it to ~57% [26] . Another study using a more advanced CNN variant reached about **68.5%** on FMA-small [13] , highlighting the challenge. With our pipeline's data augmentation and a well-chosen architecture, we might push a scratch model's accuracy into the **70–80%** range. For instance, using multiple 5-second segments per song for training (i.e. effectively 48k training samples) and an ensemble of their predictions, one team boosted a CNN's performance and reported ~65% on FMA-small [27] . We anticipate our scratch CNN, with heavy augmentation and segment ensembling, could approach the higher end of this range (perhaps **≈75%+**). However, it likely *won't match the fine-tuned model*, which benefits from far greater learned capacity.

Trade-Offs:

- **Data Efficiency:** Pretrained models shine in data-scarce regimes – they are *more data-efficient*, learning the task with fewer examples [22] . The scratch CNN needs more data or augmentation to generalize. Fine-tuning leverages prior learning ("universal" audio features like certain instrument sounds or beat patterns) so it starts from a strong baseline, whereas the scratch model must learn everything from zero.

- **Computational Cost (Training):** The custom CNN is lightweight – on a CPU it will train faster per epoch and use less memory. The AST or PANNs model, in contrast, is much larger (AST has millions of parameters, and PANNs CNN14 has 79 layers [28] [29] with around 10 million parameters). Fine-tuning the large model on CPU can be **very slow** – possibly taking hours per epoch. If GPU is unavailable, one might opt to freeze most layers or use smaller pretrained models (there are lighter variants of PANNs or DistilAST) to reduce computation. The scratch CNN might train to convergence in a few hours on CPU, whereas the pretrained could take significantly longer if fully fine-tuned. However, because the pretrained model may need fewer epochs to converge, the total training time might be somewhat mitigated – e.g., you might do 5–10 epochs of fine-tuning vs 30+ epochs for scratch. Still, **on MacBook CPU, the scratch model is more practical to iterate on**.

- **Model Size and Inference Speed:** The fine-tuned AST/PANNs model will be larger on disk and slower to run for inference. A big AST might require a dedicated GPU for real-time use. The custom CNN, being smaller, will have a fast inference on CPU and a smaller memory footprint, which is advantageous for deployment on a laptop or mobile device. If the application needs on-device processing (e.g. a mobile app classifying songs), the smaller CNN could be preferable. The larger model could be pruned or distilled to help with this, or run on a server.

- **Accuracy vs. Complexity:** If **highest accuracy** is the priority and some computational resources are available, the pretrained route is best – one can achieve substantially higher accuracy (possibly 10–20 percentage points higher in this case) [24] [26] . The trade-off is complexity: integrating a HuggingFace AST or PANNs requires more setup (ensuring the input feature extractor is correct [30] [31] , handling

the model architecture, etc.), whereas a scratch CNN can be implemented with a few Keras/PyTorch layers. Debugging and modifying the scratch model is easier for a developer, while the AST is a black-box with fixed architecture (though you can still adjust some hyperparameters). In summary, **fine-tuning yields top accuracy at the cost of speed and simplicity**, whereas **a scratch CNN is simpler and faster on CPU but likely less accurate**.

- **Overfitting and Generalization:** A large pretrained model has high capacity, but its prior training acts as a regularizer – it may not overfit as quickly to the small dataset as a similarly large random-initialized model would. In practice, we still monitor for overfitting; techniques like early stopping or fine-tuning only the last layers first help avoid fitting noise. The scratch CNN, if small, has limited capacity which can actually be an advantage to prevent overfitting (the model may act as its own regularization by being under-complex). However, if the scratch model is made too large in an attempt to boost accuracy, it could severely overfit given 6400 training clips (especially if genres have internal diversity). Augmentation and validation monitoring are crucial in both cases. Empirically, the fine-tuned model will likely achieve higher *true* generalization because it starts from features that are known to generalize across audio domains [23].

In summary, **fine-tuning a pretrained AST/PANNs is the recommended approach for maximum accuracy**, potentially reaching ~90% genre accuracy, while a **from-scratch CNN** might reach ~70–80% at best under the same conditions. The scratch model is advantageous for quick experimentation and CPU-bound environments, but the pretrained model will provide a superior classification performance given the constraints.

Deployment in a Real-World Application

Finally, consider how to use the trained genre classifier in a practical scenario. A real-world application could be a music streaming service, a music library organizer, or a recommendation system that needs **automatic genre tagging** for audio content. Here's how deployment might work:

- **Model Packaging:** After training, export the model (the CNN or fine-tuned network) in a suitable format. For example, in PyTorch save a `.pt` model, or in TensorFlow save a SavedModel/`h5`. You might also convert the model for efficiency – e.g. use ONNX or Core ML for an iOS app, or TensorRT for a server GPU. The custom CNN, being smaller, is easy to convert and embed in mobile apps. The AST model, if used, might be deployed on a server or using a specialized accelerator given its size.

- **Ingestion Pipeline:** In a live system, new audio (a song file or stream) would be fed into the same preprocessing pipeline as training. For a full song, you could take a 30s representative clip or multiple clips. The audio is loaded, resampled to 44.1kHz if needed, converted to mono, then a mel spectrogram (128 mel bands, 2048 FFT, 512 hop) is computed. This must use the **exact same settings** as training so that the model sees the same feature format.

- **Batch or Real-Time Processing:** Depending on the use case:

- *Batch mode:* A music library organizer app might process a list of audio files, compute their spectrograms, and run the model on each to tag genre. Since it's not time-critical, it can even be done on the CPU in the background.

- *Real-time:* A streaming service or radio might classify genre on the fly as music plays. In this case, one could use a sliding window (e.g. classify every 10s segment continuously). The model could output a genre probability every few seconds and one could display the predominant genre or use it for content filtering. For real-time on device, the smaller CNN is more feasible (it could potentially run in real-time on a modern CPU). The large model might need a GPU or the Apple Neural Engine (if converted to Core ML on an iPhone, for example).

- **Using the Predictions:** Once the model outputs a genre label (or a probability distribution over the 8 genres), this can be used in various ways:

- **Tagging/Database:** The genre is saved to the song's metadata. For a personal library, the user can now search or sort by these genre tags. For a streaming platform, this enriches the song's metadata in their database.
- **Recommendation System:** The genre prediction can feed into recommendation algorithms. For example, if a user often listens to a certain genre, the system can suggest other songs the model identified as that genre. Automated playlists can be generated by picking songs with matching genres [32].
- **Music Discovery:** If a platform ingests 60,000 new tracks a day (as Spotify does) [33], manual genre labeling is impossible. An automated genre classifier can instantly categorize each new track, helping editorial teams and algorithms to sort content by style. This enables **efficient, scalable curation** of massive audio streams [33].

- **Visualization:** In a UI, one could display the predicted genre label to users, or even show confidence levels (e.g. "This track is 95% likely to be Rock, 5% Pop"). This can enhance user experience by providing context for unknown songs.

- **Performance and Monitoring:** In production, it's important to monitor the classifier's confidence and performance. One could set a confidence threshold – e.g. if the model isn't at least, say, 80% confident in any genre for a track, maybe tag it as "uncertain" or send it for manual review. Continuous evaluation on new data (perhaps user feedback or known genre labels) can be used to further fine-tune or update the model. Because genres can be subjective and songs sometimes blend genres, a robust system might allow multiple genre tags per song if confidence is split.

- **Scalability:** The pipeline should be designed to handle large volumes. The spectrogram extraction is the most computationally heavy part; this can be optimized with libraries like Librosa's accelerated backend or parallel processing (extract features on multiple threads or machines). The model inference for a small CNN is fast (a matter of milliseconds per song on CPU). For the large model, one might use GPU servers to process songs in bulk. The overall system could queue audio clips for processing, compute features, run the model, and store results in a database.

- **Maintainability:** As new music trends emerge, the model might need retraining or fine-tuning on more recent data to recognize new subgenres. The modular pipeline we built makes this straightforward – you can periodically gather new genre-labeled data, update the training set, and either fine-tune the existing model further or train a new one, then deploy the updated model.

In a real application example: imagine a **music streaming service**. A new track is uploaded by an artist without a genre tag. The service backend feeds the first 30 seconds of the track through the log-mel

spectrogram extractor, then through the genre classifier model. The model predicts, say, “Hip-Hop” with 90% confidence. The system then tags the track as Hip-Hop in its database. Now the track can appear in Hip-Hop playlists, radio stations, or recommendation sections automatically, improving user experience without any manual tagging. The entire process – from audio upload to genre assignment – can happen in seconds, demonstrating the practical value of the deep learning pipeline.

Overall, the deployment involves hooking up the trained model to an audio input pipeline and integrating the outputs into the application’s logic. With the approaches compared here, one might choose the **fine-tuned model for a cloud-based solution** where accuracy is paramount and computational resources are available, or the **lightweight CNN for on-device applications** where real-time performance and low resource usage are important.

References

- Preprocessing and mel spectrogram generation [2] [5]
- Data segmentation strategies for music clips [11]
- Example augmentations for audio training [16]
- Fine-tuning vs. scratch training efficiency [22] [23]
- Fine-tuned model accuracy on genre tasks [24] [25]
- Scratch model accuracy on FMA dataset [26]
- Real-world need for automated genre tagging [33] [32]

---

[1] [6] [7] Fine-tuning a model for music classification - Hugging Face Audio Course
https://huggingface.co/learn/audio-course/en/chapter4/fine-tuning

[2] [3] [4] [5] [8] [9] [10] [18] [21] [32] Musical Genre Classification with Convolutional Neural Networks | by Leland Roberts | TDS Archive | Medium
https://medium.com/data-science/musical-genre-classification-with-convolutional-neural-networks-ff04f9601a74

[11] [15] [19] [26] [27] [33] Comparative anlysis of the novel Audio Spectrogram Transformer (AST) for FMA Genre Classification - HackMD
https://hackmd.io/@jimvos/rJltGnVvh

[12] Music genre classification with parallel convolutional neural ... - Nature
https://www.nature.com/articles/s41598-025-90619-7

[13] Music genre classification with modified residual learning and dual ...
https://pmc.ncbi.nlm.nih.gov/articles/PMC12520411/

[14] [PDF] Music Genre Classification Using Deep Learning - CS230
https://cs230.stanford.edu/files_winter_2018/projects/6936608.pdf

[16] [22] [23] [30] [31] How To Fine-Tune The Audio Spectrogram Transformer On Your Own Data | Renumics GmbH
https://renumics.com/blog/how-to-fine-tune-the-audio-spectrogram-transformer

[17] [24] [28] [29] GitHub - qiuqiangkong/panns_transfer_to_gtzan
https://github.com/qiuqiangkong/panns_transfer_to_gtzan

20 GitHub - yijerjer/music_genre_classification
https://github.com/yijerjer/music_genre_classification

25 Mel-spectrograms of four music genres, Pop, Rock, Jazz, and Classical,... | Download Scientific Diagram
https://www.researchgate.net/figure/Mel-spectrograms-of-four-music-genres-Pop-Rock-Jazz-and-Classical-in-the-GTZAN_fig1_371727208