Llama 3

- native multimodality in llama 3 was achieved using a compositional approach
- the system uses separate encoders for different modalities like images, video, and speech
- each encoder is pretrained on large datasets relevant to its modality.
    - for example, the image encoder is trained on image-text pairs, teaching the model how visual content relates to natural language
    - the speech encoder is trained in a self-supervised way by masking and reconstructing speech inputs
- they used adapters (cross-attention layers) to integrate the outputs of these encoders into the pretrained language model
- the image adapter aligns image representations with the language model through cross-attention
- similarly, the speech adapter maps speech encodings to tokens that the language model can process
- the language model itself isn't modified during these steps—just the adapters and the encoders are trained for the multimodal tasks
- the whole setup is designed to handle images, videos, and speech without needing to retrain the entire model, and it shows competitive performance across tasks related to these modalities

How does cross attention work?

- allows one set of inputs (e.g., text) to focus on relevant parts of another set of inputs (e.g., image or speech embeddings).
- uses the standard transformer mechanism where:
    - queries (Q) come from the target modality (e.g., text).
    - keys (K) and values (V) come from the source modality (e.g., image or speech).
- queries, keys, and values are calculated using attention heads in the transformer.
- the model matches queries (text) with relevant keys and values (image or speech embeddings).
- based on the attention scores, it extracts important info from the source modality and integrates it into the target modality's understanding.
- fusing modalities: bridges different modalities (e.g., text + images) by attending to relevant details from one while processing another.
- contextual relevance: text can ask (via queries) for the most relevant details from image/speech embeddings (keys/values) to generate or understand text-based outputs with richer, multimodal context.
- maintaining separate encoders: allows independent encoders for different modalities (image/speech/text) while blending their outputs at key points in the transformer using cross-attention.
- dynamic focus: enables selective attention to the most useful parts of non-text modalities (e.g., specific objects in an image) while processing or generating language.
- efficient integration: integrates extra modalities without modifying the core language model itself, just by adding cross-attention layers to selectively bring in the relevant info from other modalities.

Llava

- llava uses the CLIP model to encode images and then projects that into the language model.
- it's very modular, with the vision encoder (clip) feeding into the text-based decoder, but it’s still using a tokenizer for the text part
- In 1.5, they used an MLP projection which improved performance

Gpt 4o

- gpt-4o doesn’t use a separate dedicated image encoder like clip or a vision transformer (vit) plugged in externally. it does embed the inputs, but the embedding process is unified into the main model architecture, not bolted on as a separate module.
- image patching: the image is split into small patches, similar to what you’d see in vision transformers (ViT).
    - each of these patches is treated like a token, similar to how words or subwords are tokens in text.
    - think of each patch as a mini-image that's small enough to be manageable but still retains some information about the original image.
- linear projection: these image patches are passed through a linear layer (a matrix multiplication, essentially), which projects them into the same vector space as the text tokens.
    - this linear projection is what's doing the "embedding" for the image patches
    - it's not a separate encoder in the traditional sense (like a whole other network).
    - it's more like a direct transformation step built right into the model.
- positional embeddings: just like text tokens get positional embeddings to maintain word order, image patches also get positional embeddings to encode their spatial layout within the image.
    - this helps the model know where each patch fits in relation to the whole image (e.g., top-left, center, etc.).
- once the image patches are transformed into vectors with the linear projection, they’re treated exactly like text tokens.
- the transformer model doesn't care if a token came from an image or from text—it processes them all in the same way, through the same transformer layers.

Next steps: project raw inputs into embedding space

- Splitting raw waveforms into chunks
- Preserves temporal info – don’t need positional encodings

```
import torch
import torch.nn as nn

# Define a simple Decoder Block with self-attention
class DecoderBlock(nn.Module):
    def __init__(self, embedding_dim, num_heads, ff_dim):
        super(DecoderBlock, self).__init__()
        self.self_attention = nn.MultiheadAttention(embedding_dim, num_heads)
        self.norm1 = nn.LayerNorm(embedding_dim)
        self.ffn = nn.Sequential(
            nn.Linear(embedding_dim, ff_dim),
            nn.ReLU(),
            nn.Linear(ff_dim, embedding_dim)
        )
        self.norm2 = nn.LayerNorm(embedding_dim)

    def forward(self, x):
        # Self-attention over the input (previous output, or chunks of waveform)
        attn_output, _ = self.self_attention(x, x, x)
        x = self.norm1(x + attn_output)  # Add & Norm

        # Feed-forward network
        ffn_output = self.ffn(x)
        x = self.norm2(x + ffn_output)  # Add & Norm

        return x

# Define a Decoder-only Transformer that accepts raw waveform chunks
class WaveformDecoderTransformer(nn.Module):
    def __init__(self, embedding_dim, num_heads, ff_dim, num_layers):
        super(WaveformDecoderTransformer, self).__init__()
        self.embedding_dim = embedding_dim

        # Linear projection to embed raw waveform chunks
        self.linear_projection = nn.Linear(input_chunk_size, embedding_dim)

        # Stack of decoder blocks
        self.decoder_blocks = nn.ModuleList([
            DecoderBlock(embedding_dim, num_heads, ff_dim)
            for _ in range(num_layers)
        ])

        # Output layer to map back to the waveform domain, if necessary
        self.output_layer = nn.Linear(embedding_dim, output_dim)  # Adjust for your task
    def forward(self, waveform_chunks):
        # Project raw waveform chunks into the embedding space
        embedded_chunks = self.linear_projection(waveform_chunks)

        # Prepare input for the decoder (make sure it's in the right format for attention layers)
        # [sequence_length, batch_size, embedding_dim]
        embedded_chunks = embedded_chunks.permute(1, 0, 2)  # Transpose to fit attention input

        # Pass through all decoder layers
        for decoder_block in self.decoder_blocks:
            embedded_chunks = decoder_block(embedded_chunks)

        # Optionally map to output space (e.g., back to waveform or logits for tasks like speech
recognition)
        output = self.output_layer(embedded_chunks.permute(1, 0, 2))  # Reshape back to original
format

        return output

# Example usage
# Assume raw_waveform is a 2D tensor with shape [batch_size, waveform_length]
# We'll split it into smaller chunks of size `input_chunk_size`
raw_waveform = torch.randn(batch_size, waveform_length)  # Example random waveform input

# Function to split raw waveform into chunks
def split_waveform_into_chunks(waveform, chunk_size):
    num_chunks = waveform.shape[1] // chunk_size
    waveform_chunks = waveform[:, :num_chunks * chunk_size].reshape(batch_size,
num_chunks, chunk_size)
    return waveform_chunks

# Hyperparameters
input_chunk_size = 512   # Size of each chunk of raw waveform
embedding_dim = 256      # Dimension of embedding space
num_heads = 8            # Number of attention heads
ff_dim = 512             # Feed-forward network dimension
num_layers = 6           # Number of decoder layers
output_dim = 512         # Output dimension (e.g., might be same as chunk size)

# Initialize the model
model = WaveformDecoderTransformer(embedding_dim, num_heads, ff_dim, num_layers)

# Split raw waveform into chunks
```

```
waveform_chunks = split_waveform_into_chunks(raw_waveform, input_chunk_size)

# Forward pass through the model
output = model(waveform_chunks)  # The output will be projected back to the waveform
domain, or any other task-specific output
```