Complete Explanation of models.py

Overview

This file implements **4 progressively complex models** for audio-to-text captioning, from simple baseline to state-of-the-art transformer.

🎯 File Structure

```
models.py
├── Sampling Utilities (prevents repetitive captions)
├── Level 1: BaselineModel (simplest)
├── Level 2: ImprovedBaselineModel (deeper)
├── Level 3: AttentionModel (focuses on audio parts)
├── Level 4: TransformerModel (most advanced)
└── Model Factory (easy creation)
```

🔧 Part 1: Sampling Utilities

`sample_with_temperature()`

What it does: Generates diverse text by adding controlled randomness

```python
def sample_with_temperature(logits, temperature=1.0, top_k=0, top_p=0.9):
```

Instead of:

```python
# OLD (greedy - always picks most likely word)
next_word = logits.argmax()  # "person" every time
```

Does:

```python
# NEW (sampling - picks from distribution)
next_word = sample_from_distribution()  # "person", "engine", "bird", etc.
```

Three mechanisms:

1. **Temperature**: Controls randomness

```python
logits = logits / temperature
# temperature = 0.1 → very focused (almost greedy)
# temperature = 1.0 → normal distribution
# temperature = 2.0 → very random
```

2. **Top-k filtering**: Only consider k most likely words

```python
# Instead of all 4374 words, only look at top 50
# Removes weird/rare words
```

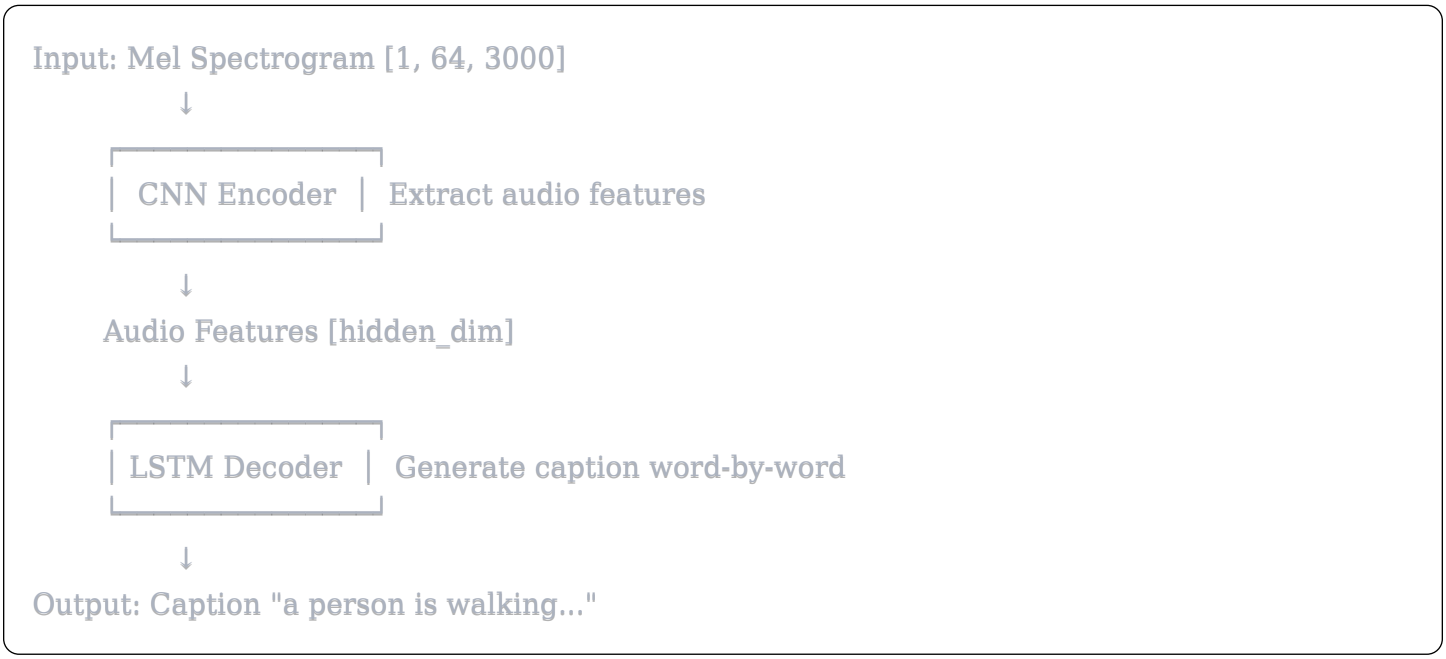
3. **Top-p (nucleus) sampling**: Consider smallest set that covers p% probability

```python
# Adaptive: sometimes uses 5 words, sometimes 50
# Keeps 90% probability mass
```

Why this matters: Prevents "a person is walking" repetition!

📊 Part 2: Level 1 - BaselineModel

Architecture Overview



Key Components

1. CNN Encoder

```python
self.encoder = nn.Sequential(
    nn.Conv2d(1, 64, kernel_size=3, padding=1),
    nn.AvgPool2d(2, 2),  # KEY: AvgPool NOT MaxPool
    ...
)
```

Why AvgPool instead of MaxPool?

- **MaxPool:** Takes maximum value → loses 75% of information
- **AvgPool:** Takes average → preserves all information smoothly

Processing flow:

```
Input:  [batch, 1, 64, 3000]  (mel spectrogram)
Conv1:  [batch, 64, 64, 3000]
Pool1:  [batch, 64, 32, 1500] (reduced by 2x)
Conv2:  [batch, 128, 32, 1500]
Pool2:  [batch, 128, 16, 750]
Conv3:  [batch, 256, 16, 750]
Pool3:  [batch, 256, 8, 375]
```

Then reshapes to preserve temporal info:

```python
# Keep time dimension (375 timesteps)
features.permute(0, 3, 1, 2)  # Move time to front
# Result: [batch, 375, 256*8]
# Now we have 375 timesteps, each with 2048 features
```

2. LSTM Decoder

```python
self.decoder_lstm = nn.LSTM(
    embed_dim,      # Input: word embedding
    hidden_dim,     # Hidden state size
    num_layers,     # Stack of 2 LSTMs
    batch_first=True
)
```

How it generates:

```
Step 1: Input: <sos> + audio_context → Output: "a"
Step 2: Input: "a" + audio_context → Output: "person"
Step 3: Input: "person" + audio_context → Output: "is"
...
```

3. Generation Process

```
python

def generate(self, mel, max_len=30, sos_idx=1, eos_idx=2,
             temperature=1.0, top_k=0, top_p=0.9):
```

Step-by-step:

1. Encode audio → get audio context vector
2. Initialize LSTM with audio context
3. Start with `<sos>` token
4. For each position (max 30 words):
   - Embed current word
   - Pass through LSTM
   - Get logits (scores for each word)
   - **Sample** (not argmax!) using temperature
   - Append to sequence
5. Stop when `<eos>` generated or max_len reached

📈 Part 3: Level 2 - ImprovedBaselineModel

What's Different?

```
BaselineModel → ImprovedBaselineModel

Changes:
1. Deeper CNN (4 layers instead of 3)
2. Bidirectional LSTM for audio encoding
3. Audio context concatenated with each word embedding
```

Key Improvement: Bidirectional LSTM

```python
self.temporal_lstm = nn.LSTM(
    512 * 4,
    hidden_dim // 2,
    bidirectional=True,  # ← KEY!
    batch_first=True
)
```

Why bidirectional?

Unidirectional (baseline):

```
Audio: [t1] → [t2] → [t3] → [t4] → [t5]
       ↓      ↓      ↓      ↓      ↓
      h1 → h2 → h3 → h4 → h5

At t3, only knows about t1, t2, t3
```

Bidirectional (improved):

```
Audio: [t1] ← [t2] ← [t3] ← [t4] ← [t5]  (backward)
       ↓      ↓      ↓      ↓      ↓
      h1 → h2 → h3 → h4 → h5       (forward)

At t3, knows about ALL timesteps t1-t5
```

Audio Context at Every Step

```python
# Embed captions
embedded = self.embedding(captions)  # [batch, seq_len, embed_dim]

# Repeat audio context for each word
audio_context_expanded = audio_context.unsqueeze(1).repeat(1, seq_len, 1)

# Concatenate
decoder_input = torch.cat([embedded, audio_context_expanded], dim=-1)
```

Why this helps:

- Forces model to **always consider audio** when generating each word
- Can't "forget" what the audio sounds like
- Word generation is conditioned on both previous words AND audio

---

🎯 Part 4: Level 3 - AttentionModel

The Big Innovation: Attention Mechanism

Problem with previous models:

- Audio encoded to single vector
- Same vector used for all words
- Can't focus on different audio parts for different words

Solution: Attention

- Keep audio as **sequence** of features (not single vector)
- At each generation step, **attend** to relevant parts

How Attention Works

Bahdanau Attention

```python
class BahdanauAttention(nn.Module):
    def forward(self, encoder_outputs, decoder_hidden):
        # encoder_outputs: (batch, time_steps, hidden_dim)
        # decoder_hidden: (batch, hidden_dim)

        # 1. Compute attention scores
        score = V(tanh(W_encoder(encoder) + W_decoder(decoder)))

        # 2. Softmax to get weights
        attention_weights = softmax(score)  # Sum to 1

        # 3. Weighted sum of encoder outputs
        context = sum(attention_weights * encoder_outputs)

        return context
```

Visual Example:

Generating caption for engine sound:

```
Audio features over time:
[quiet] [revving] [loud] [sustain] [fadeout]
  ↓       ↓       ↓       ↓         ↓
 0.1     0.5     0.3     0.1       0.0    ← Attention when generating "revs"

 0.05    0.15    0.6     0.15      0.05   ← Attention when generating "loudly"
```

At each word generation:

- Model asks: "Which part of audio is relevant NOW?"
- Attention weights = where to focus
- Context vector = weighted combination of audio features

Generation with Attention

python

```python
for t in range(max_len):
    # 1. Compute attention (WHERE to look in audio)
    context, attention_weights = self.attention(encoder_outputs, h[-1])

    # 2. Use attended context for generation
    lstm_input = torch.cat([embedded, context], dim=-1)
    lstm_out, (h, c) = self.decoder_lstm(lstm_input, (h, c))

    # 3. Generate next word
    output = torch.cat([lstm_out, context], dim=-1)
    logits = self.output_projection(output)
```

Why this is powerful:

- Word "engine" → attends to low-frequency rumble
- Word "loudly" → attends to high-energy regions
- Word "intermittent" → attends to gaps/pauses
- Model learns what to focus on automatically!

🚀 Part 5: Level 4 - TransformerModel

The Most Advanced Architecture

Replaces:

- CNN + RNN → CNN + Transformer

Key difference:

- RNN processes sequentially: t1 → t2 → t3 → ...
- Transformer processes in parallel: all timesteps at once!

Components

1. Positional Encoding

```python
class PositionalEncoding(nn.Module):
```

Problem: Transformers have no sense of order **Solution:** Add position information

```python
# For position 0:
PE[0] = [sin(0/10000^0), cos(0/10000^0), sin(0/10000^1), ...]

# For position 1:
PE[1] = [sin(1/10000^0), cos(1/10000^0), sin(1/10000^1), ...]

# Add to embeddings
output = embedding + positional_encoding
```

2. Transformer Architecture

```
Audio Processing:
CNN → [batch, 187, 2048]
  ↓
Project → [batch, 187, d_model=512]
  ↓
+ Positional Encoding
  ↓
Transformer Encoder (3 layers)
  ├─ Multi-Head Self-Attention
  │  (audio attends to itself)
  ├─ Feed-Forward Network
  └─ Layer Norm + Residual
  ↓
Audio Memory: [batch, 187, 512]

Caption Generation:
Embed words → [batch, seq_len, d_model=512]
  ↓
+ Positional Encoding
  ↓
Transformer Decoder (3 layers)
  ├─ Masked Self-Attention
  │  (can't see future words)
  ├─ Cross-Attention
  │  (attends to audio memory)
  ├─ Feed-Forward Network
  └─ Layer Norm + Residual
  ↓
Logits: [batch, seq_len, vocab_size]
```

3. Multi-Head Attention

Instead of single attention:

```
Single: One way to attend to audio
```

Multi-head (8 heads):

```
Head 1: Focuses on rhythm patterns
Head 2: Focuses on pitch
Head 3: Focuses on loudness
Head 4: Focuses on timbre
...
Head 8: Focuses on temporal structure

Combine all → Rich representation
```

4. Causal Masking

python

```python
def generate_square_subsequent_mask(self, sz, device):
    mask = torch.triu(torch.ones(sz, sz), diagonal=1)
    mask = mask.masked_fill(mask == 1, float('-inf'))
    return mask
```

Why needed?

During training:

```
Input: "a person is walking"
Target: "person is walking on"

Without mask:
- Predicting "person" → can see "is", "walking", "on" (CHEATING!)

With mask:
- Predicting "person" → can only see "a" (CORRECT!)
```

Mask looks like:

```
      a    person   is    walking
a     0    -inf    -inf   -inf     ← Can only see "a"
person 0    0       -inf   -inf     ← Can see "a", "person"
is    0     0       0      -inf     ← Can see "a", "person", "is"
walking 0   0       0      0        ← Can see all
```

Why Transformers are Powerful

Advantages:

1. **Parallel processing**: Much faster than RNN
2. **Long-range dependencies**: Can relate words far apart
3. **Attention everywhere**: Every position attends to every other
4. **State-of-the-art**: Best results on most tasks

Disadvantages:

1. **More parameters**: Needs more data
2. **More compute**: Requires better GPU
3. **Harder to train**: More hyperparameters to tune

🏭 Part 6: Model Factory

```python
def create_model(model_type, vocab_size, **kwargs):
    models = {
        'baseline': BaselineModel,
        'improved_baseline': ImprovedBaselineModel,
        'attention': AttentionModel,
        'transformer': TransformerModel
    }
    return models[model_type](vocab_size, **kwargs)
```

Usage:

```python
# Easy model creation
model = create_model('attention', vocab_size=4374, hidden_dim=512)

# Or manually
model = AttentionModel(vocab_size=4374, hidden_dim=512)
```

📊 Model Comparison

Parameters & Complexity

Model	Params	Speed	Quality	Best For
Baseline	~5M	Fast	Basic	Testing pipeline
Improved	~8M	Medium	Better	Quick experiments
Attention	~10M	Medium	Good	Production baseline
Transformer	~15M	Slow	Best	Final model

When to Use Each

Baseline:

- ✅ First experiments
- ✅ Debugging data pipeline
- ✅ Quick prototyping
- ❌ Production use

Improved Baseline:

- ✅ Better results without too much complexity
- ✅ Good balance of speed/quality
- ✅ When compute is limited
- ❌ State-of-the-art results

Attention:

- ✅ Industry standard
- ✅ Interpretable (can visualize attention)
- ✅ Good performance
- ✅ Recommended starting point

Transformer:

- ✅ Best performance (if enough data)
- ✅ State-of-the-art architecture
- ❌ Needs more data (Clotho might be too small)
- ❌ Slower to train

🔑 Key Design Decisions

1. Why AvgPool instead of MaxPool?

Throughout all models:

```python
nn.AvgPool2d(2, 2)  # NOT MaxPool2d
```

Reason:

- **MaxPool**: Throws away 75% of information
- **AvgPool**: Smoothly combines all information
- Audio has subtle features → need to preserve everything

2. Why Keep Temporal Dimension?

Instead of:

```python
# BAD: Single vector
features = cnn(mel)  # [batch, 512, 4, 187]
features = features.mean(dim=(2,3))  # [batch, 512] ← Lost all time info!
```

Do:

```python
# GOOD: Sequence of vectors
features = cnn(mel)  # [batch, 512, 4, 187]
features = reshape(features)  # [batch, 187, 2048] ← 187 timesteps preserved!
```

Why: Audio events happen over time! Need to know WHEN things happen.

3. Why Audio Context at Every Step?

Improved Baseline and Attention:

```python
# Every word generation uses audio
decoder_input = torch.cat([word_embedding, audio_context], dim=-1)
```

Why: Forces model to consider audio when generating each word, not just first word.

4. Why Sampling Instead of Argmax?

All models use:

```python
next_token = sample_with_temperature(logits, temperature=0.8, top_p=0.9)
# NOT: next_token = logits.argmax()
```

Why: Prevents repetitive "a person is walking..." captions.

💻 Usage Examples

Training

```python
from models import create_model

# Create model
model = create_model('attention', vocab_size=4374, hidden_dim=512)

# Forward pass (training)
mel = torch.randn(32, 1, 64, 3000)  # Batch of 32 audio clips
captions = torch.randint(0, 4374, (32, 20))  # Batch of captions

logits = model(mel, captions)  # [32, 20, 4374]

# Compute loss
loss = criterion(logits.reshape(-1, 4374), target.reshape(-1))
```

Generation

python

```python
# Generate caption
mel = torch.randn(1, 1, 64, 3000)  # Single audio

generated = model.generate(
    mel,
    max_len=30,
    sos_idx=vocab['<sos>'],
    eos_idx=vocab['<eos>'],
    temperature=0.8,  # Some randomness
    top_p=0.9         # Nucleus sampling
)

# Decode to text
caption = decode_ids_to_text(generated[0], vocab)
print(caption)
```

🎓 Learning Progression

Week 1: Baseline

- Understand CNN → LSTM flow
- Get familiar with audio processing
- Establish baseline metrics

Week 2: Improved Baseline

- Learn about bidirectional LSTMs
- Understand audio context concatenation
- See quality improvements

Week 3: Attention

- Grasp attention mechanism
- Visualize attention weights
- Significant quality boost

Week 4: Transformer

- Understand self-attention
- Learn about positional encoding
- Push to state-of-the-art

Summary

This `models.py` file provides:

1. **4 model levels** from simple to advanced
2. **Sampling utilities** to prevent repetition
3. **Consistent interface** across all models
4. **Best practices** built-in (AvgPool, temporal preservation, attention)
5. **Production-ready code** with proper documentation

Each model builds on the previous one, allowing systematic experimentation to find the best architecture for your specific needs!