MULTIMODAL MEDICAL MODELS

Text + Image

Radiology reports with X-rays, CT scans, MRI

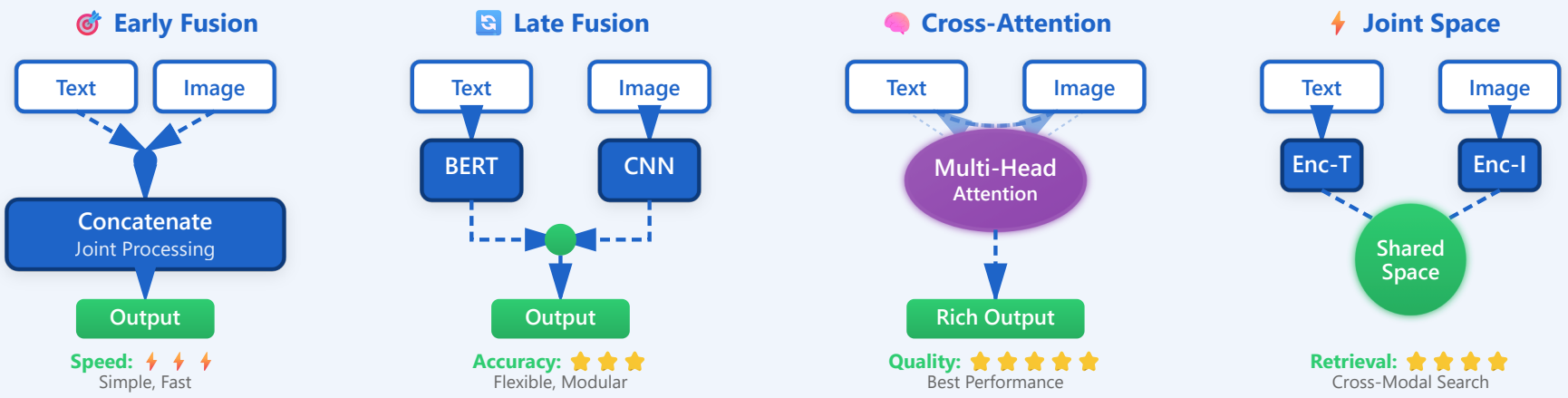
Text + Signal

ECG, EEG, vital signs waveform analysis

Text + Video

Surgical procedures, endoscopy, ultrasound

Fusion Strategies Comparison



1 Early Fusion (Input-Level Fusion)

Combine raw inputs before any processing - the simplest approach

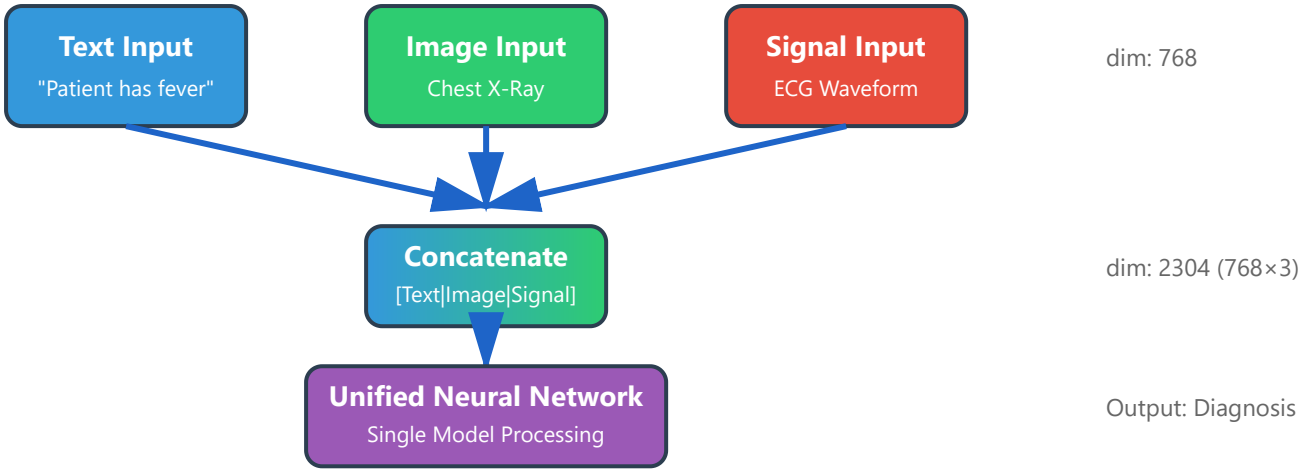
✅ Advantages

- Computationally efficient - single model processes everything
- Simple architecture - easy to implement and debug
- Low latency - fastest inference time
- Learns joint representations from the start
- Minimal memory overhead

❌ Disadvantages

- May lose modality-specific features
- Harder to pretrain on unimodal data
- Less flexible - can't swap out modality encoders
- Potential for one modality to dominate
- Difficult to handle missing modalities

🔍 Early Fusion Architecture Example



🏥 Medical Application Example

Scenario: Pneumonia Detection System

A hospital implements an early fusion model that takes a chest X-ray image and concatenates it with patient vital signs (temperature, heart rate, oxygen saturation) and clinical notes. All inputs are combined into a single vector before being fed into one unified deep learning model. This approach achieves real-time inference (< 100ms) which is crucial for emergency room triage, though it may miss subtle correlations between modalities that later fusion methods could capture.

```
# Early Fusion Implementation Example import torch import torch.nn as nn class EarlyFusionModel(nn.Module): def __init__(self): super().__init__() # Concatenate all inputs at the beginning self.fusion_layer = nn.Linear(text_dim + image_dim + signal_dim, 512) self.classifier = nn.Sequential( nn.ReLU(), nn.Linear(512, 256), nn.ReLU(), nn.Linear(256, num_classes) ) def forward(self, text, image, signal): # Immediate concatenation combined = torch.cat([text, image, signal], dim=-1) fused = self.fusion_layer(combined) output = self.classifier(fused) return output
```

2 Late Fusion (Decision-Level Fusion)

Process each modality independently, then combine predictions

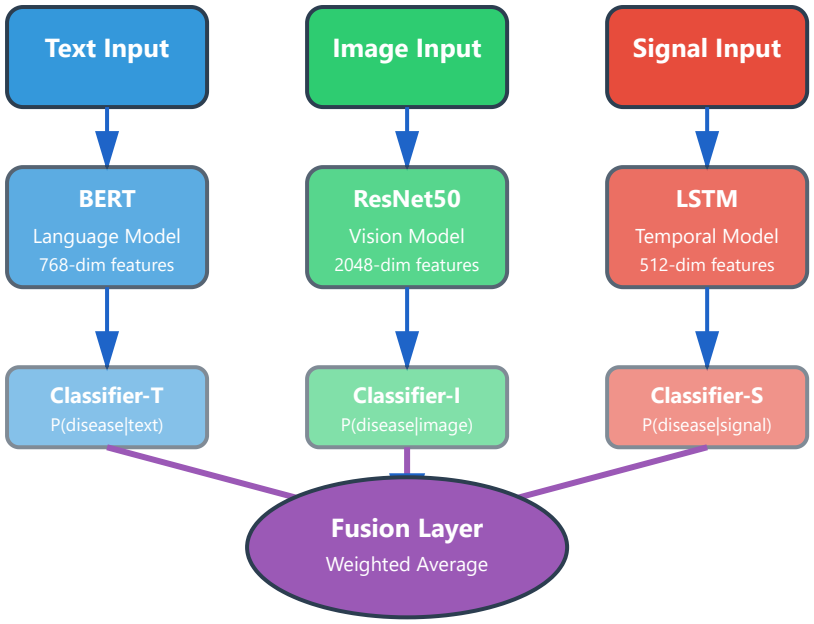
✅ Advantages

- Modular design - easy to upgrade individual components
- Can leverage pretrained models for each modality
- Handles missing modalities gracefully
- Each modality can use specialized architectures
- Easier to interpret - can see each modality's contribution

❌ Disadvantages

- Higher computational cost - multiple models running
- May miss early interaction patterns between modalities
- Requires careful fusion strategy design
- More complex training pipeline
- Higher memory requirements

🔍 Late Fusion Architecture Example



Fusion Strategies:

- Average: $P = (P_1+P_2+P_3)/3$
- Weighted: $P = w_1P_1+w_2P_2+w_3P_3$
- Max: $P = \max(P_1, P_2, P_3)$
- Learned: $MLP([P_1, P_2, P_3])$

🏥 Medical Application Example

Scenario: Cardiac Risk Assessment

A cardiology department uses late fusion for comprehensive heart disease diagnosis. Three specialized models run independently: (1) a BERT model analyzes clinical notes and patient history, (2) a ResNet analyzes echocardiogram images, and (3) an LSTM processes ECG time series. Each model provides a risk score, which are then combined using learned weights. This allows the system to work even when one modality is missing, and doctors can see which modality contributed most to the final decision.

```
# Late Fusion Implementation Example class LateFusionModel(nn.Module): def __init__(self): super().__init__() # Separate encoders for each modality self.text_encoder = BERTEncoder() self.image_encoder = ResNet50() self.signal_encoder = LSTMEncoder() # Individual classifiers self.text_classifier = nn.Linear(768, num_classes) self.image_classifier = nn.Linear(2048, num_classes) self.signal_classifier = nn.Linear(512, num_classes) # Learned fusion weights self.fusion_weights = nn.Parameter(torch.ones(3)) def forward(self, text, image, signal): # Process each modality independently text_features = self.text_encoder(text) image_features = self.image_encoder(image) signal_features = self.signal_encoder(signal) # Get predictions from each modality
```

```
text_pred = self.text_classifier(text_features) image_pred = self.image_classifier(image_features) signal_pred =
self.signal_classifier(signal_features) # Weighted fusion of predictions weights = F.softmax(self.fusion_weights,
dim=0) final_pred = (weights[0] * text_pred + weights[1] * image_pred + weights[2] * signal_pred) return
final_pred
```

3 Cross-Attention Fusion

Enable modalities to attend to each other dynamically

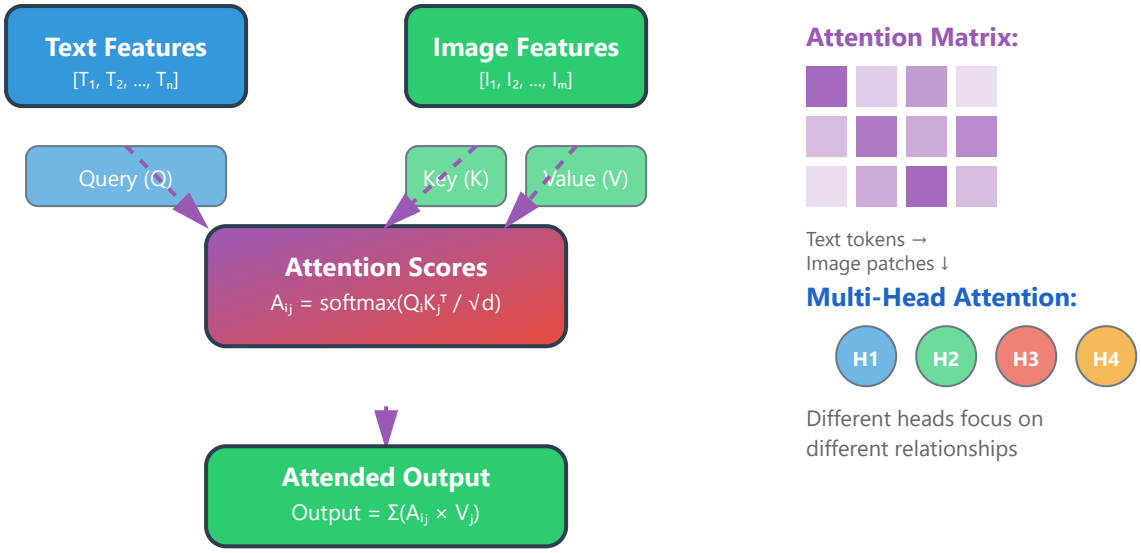
✅ Advantages

- Captures fine-grained inter-modal relationships
- State-of-the-art performance on complex tasks
- Interpretable attention weights show modal interactions
- Flexible - can attend to relevant parts of each modality
- Works well with transformer architectures

❌ Disadvantages

- Computationally expensive - quadratic complexity
- Requires more training data to learn attention patterns
- Longer training time
- Higher memory consumption
- May require careful regularization to prevent overfitting

🔍 Cross-Attention Mechanism Example



Key Equations:

1. $\text{Attention}(Q, K, V) = \text{softmax}(QK^T/\sqrt{d_k}) \times V$
2. $\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1,...,\text{head}_h)W^O$ where $\text{head}_i = \text{Attention}(QW_i^\varphi, KW_i^k, VW_i^v)$

🏥 Medical Application Example

Scenario: Pathology Report Generation

An AI pathology assistant uses cross-attention to generate diagnostic reports from histopathology images. The model learns to attend to specific regions in the microscopy image (like cellular structures) when generating corresponding text descriptions. For instance, when writing "mitotic figures are abundant," the attention mechanism focuses on the relevant cellular regions in the image. This creates interpretable AI where pathologists can see exactly which image regions influenced each part of the generated report, improving trust and adoption in clinical settings.

```
# Cross-Attention Implementation Example class CrossAttentionFusion(nn.Module): def __init__(self, d_model=512, n_heads=8): super().__init__() self.text_encoder = TextEncoder() self.image_encoder = ImageEncoder() # Multi-head
```

```
cross-attention self.cross_attention = nn.MultiheadAttention( embed_dim=d_model, num_heads=n_heads,
batch_first=True ) self.norm = nn.LayerNorm(d_model) self.classifier = nn.Linear(d_model, num_classes) def
forward(self, text, image): # Encode each modality text_features = self.text_encoder(text) # [B, seq_len, d_model]
image_features = self.image_encoder(image) # [B, num_patches, d_model] # Cross-attention: text attends to image
attended_output, attention_weights = self.cross_attention( query=text_features, key=image_features,
value=image_features ) # Residual connection and normalization output = self.norm(text_features + attended_output)
# Classification pooled = output.mean(dim=1) prediction = self.classifier(pooled) return prediction,
attention_weights # Return weights for visualization
```

4 Joint Embedding Space Fusion

Map different modalities into a shared semantic space

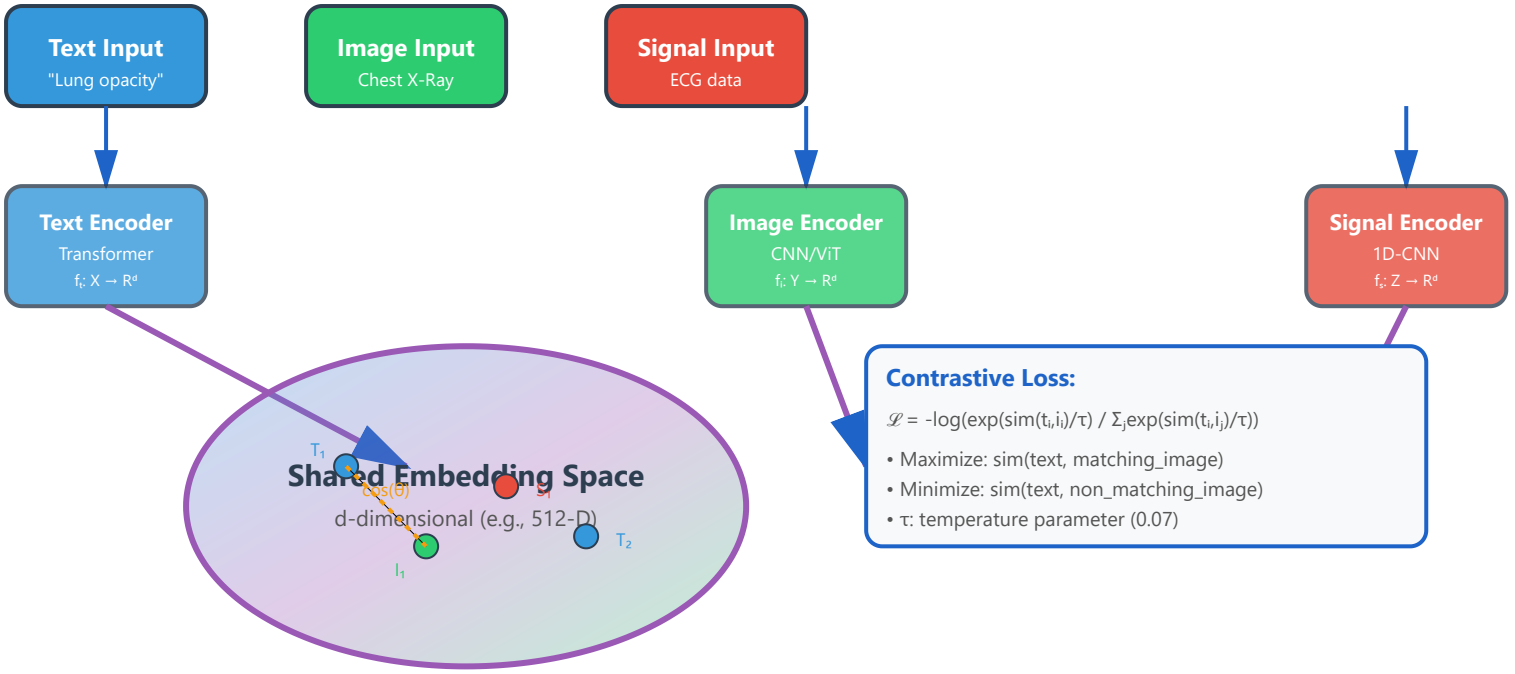
✅ Advantages

- Enables cross-modal retrieval (text → image, image → text)
- Semantically meaningful representations
- Zero-shot capabilities for unseen combinations
- Efficient similarity search across modalities
- Works well with contrastive learning

❌ Disadvantages

- Requires large paired datasets for training
- May lose modality-specific information
- Challenging to optimize - needs careful loss design
- Embedding space dimensionality choice is critical
- Can struggle with fine-grained distinctions

🔍 Joint Embedding Space Architecture



Applications:

🔍 Search X-rays by text description → cos_similarity(embed_text, embed_image)
🏷️ Zero-shot classification → find closest text label in embedding space

🩻 Medical Application Example

Scenario: Medical Image Retrieval System

A hospital implements a CLIP-style model trained on 100,000 radiology report-image pairs. Radiologists can now search the entire image database using natural language: "show me chest X-rays with bilateral infiltrates and cardiomegaly." The system maps both the text query and all stored images into the same 512-dimensional embedding space, then retrieves images with highest cosine similarity. This revolutionizes clinical workflow, reducing search time from minutes to seconds and helping doctors find relevant prior cases for comparison and learning.

```
# Joint Embedding Space Implementation (CLIP-style) class JointEmbeddingModel(nn.Module): def __init__(self, embed_dim=512): super().__init__() self.text_encoder = TextTransformer(output_dim=embed_dim) self.image_encoder = VisionTransformer(output_dim=embed_dim) # Temperature parameter for scaling self.temperature = nn.Parameter(torch.ones([]) * np.log(1 / 0.07)) def forward(self, text, image): # Project to shared embedding space text_features = self.text_encoder(text) # [B, embed_dim] image_features = self.image_encoder(image) # [B, embed_dim] # Normalize features text_features = F.normalize(text_features, dim=-1) image_features = F.normalize(image_features, dim=-1) return text_features, image_features def contrastive_loss(self, text_features, image_features): # Cosine similarity as logits logits = (text_features @ image_features.T) * self.temperature.exp() # Symmetric cross-entropy loss labels = torch.arange(len(logits)).to(logits.device) loss_t = F.cross_entropy(logits, labels) loss_i = F.cross_entropy(logits.T, labels) return (loss_t + loss_i) / 2 def retrieve(self, query_text, image_database): """Retrieve most similar images for a text query""" query_embed = self.text_encoder(query_text) query_embed = F.normalize(query_embed, dim=-1) # Compute similarities db_embeds = torch.stack([self.image_encoder(img) for img in image_database]) db_embeds = F.normalize(db_embeds, dim=-1) similarities = query_embed @ db_embeds.T top_k_indices = similarities.topk(k=5).indices return top_k_indices
```

📊 Comprehensive Comparison

Criterion	Early Fusion	Late Fusion	Cross-Attention	Joint Embedding
Computational Cost	Low ⚡⚡⚡	Medium ⚡⚡	High ⚡	Medium ⚡⚡
Performance	Medium ⭐⭐	Good ⭐⭐⭐	Excellent ⭐⭐⭐⭐⭐	Very Good ⭐⭐⭐⭐
Interpretability	Low	High	Very High	Medium
Missing Modality	Poor	Excellent	Moderate	Good
Training Data	Moderate	Moderate	Large	Very Large

Criterion	Early Fusion	Late Fusion	Cross-Attention	Joint Embedding
Modularity	Low	Very High	Medium	Medium
Cross-Modal Retrieval	No	No	Limited	Excellent
Best Use Case	Real-time, resource-constrained	Modular systems, missing data	High accuracy critical tasks	Search, retrieval, zero-shot
Medical Example	ER triage systems	Multi-source diagnosis	Report generation	Image database search

💡 Key Takeaways & Best Practices

🎯 Choosing the Right Fusion Strategy

- **Early Fusion** when you need speed and have limited compute resources
- **Late Fusion** when you have pre-trained models for each modality and need flexibility
- **Cross-Attention** when accuracy is paramount and you have sufficient computational resources
- **Joint Embedding** when you need cross-modal search or zero-shot capabilities

🔬 Practical Implementation Tips

✓ Do's

- Start simple (early/late fusion) before trying complex methods
- Normalize features before fusion
- Use appropriate data augmentation for each modality
- Monitor individual modality performance
- Implement ablation studies to validate fusion benefit

✗ Don'ts

- Don't ignore modality imbalance in training
- Don't assume more complex = better performance
- Don't forget to handle missing modalities at inference
- Don't neglect computational constraints in deployment
- Don't skip validation with domain experts