# Automated Clinical Note Classification with Self-Attention

*MedScribe Analytics -- Reducing Diagnostic Coding Errors in Hospital Documentation*

---

## 1. Industry Context and Business Problem

### The Clinical Documentation Crisis

Every year, hospitals in the United States process over 1.2 billion clinical encounters, each generating unstructured text in the form of physician notes, discharge summaries, and progress reports. These documents must be mapped to standardized diagnostic codes (ICD-10) for billing, quality reporting, and epidemiological tracking. Today, this mapping is performed by a workforce of approximately 300,000 certified medical coders, and the process is slow, expensive, and error-prone.

**MedScribe Analytics** is a health-tech company that provides automated clinical documentation intelligence to mid-size hospital networks (200-500 bed facilities). Their core product ingests free-text clinical notes and predicts the appropriate ICD-10 diagnostic codes.

### The Problem

MedScribe's current NLP pipeline uses a Bi-LSTM model trained on 2.3 million annotated clinical notes. While the model achieves 78% top-1 accuracy on common diagnoses, it suffers from three critical failure modes:

| Failure Mode | Example | Impact |
|---|---|---|
| **Long-range dependency loss** | A note mentions "type 2 diabetes" in paragraph 1 and "insulin-dependent" in paragraph 4. The LSTM forgets the diabetes mention by paragraph 4. | Miscodes 12% of multi-paragraph notes |
| **Negation mishandling** | "Patient does **not** have pneumonia" is coded as pneumonia because the negation cue ("not") is several words away from the diagnosis. | 8% false positive rate on negated diagnoses |
| **Multi-label confusion** | A note describing both "congestive heart failure" and "chronic kidney disease" fails to assign both codes because the model compresses the entire note into a single vector. | Misses 23% of secondary diagnoses |

These failures cost MedScribe's hospital clients an estimated \$2.4 million per year in denied insurance claims, compliance penalties, and manual re-coding labor. The VP of Engineering has

determined that the root cause is the **sequential bottleneck** of the LSTM architecture, and has authorized a migration to a **self-attention-based model**.

### Success Criteria

| Metric | Current (LSTM) | Target (Attention) |
|---|---|---|
| Top-1 accuracy (common codes) | 78% | 88%+ |
| Negation detection F1 | 0.72 | 0.90+ |
| Multi-label recall | 0.65 | 0.82+ |
| Inference latency (per note) | 45ms | < 60ms |
| Training time (full dataset) | 8 hours | < 12 hours |

---

## 2. Technical Problem Formulation

### 2.1 Problem Statement

Given a clinical note $x = (x_1, x_2, \ldots, x_n)$ consisting of $n$ tokens, predict a set of ICD-10 diagnostic codes $\{c_1, c_2, \ldots, c_m\}$ from a label space of 1,200 common codes.

This is a **multi-label classification** problem where each note can have 1-8 associated codes (mean: 3.2 codes per note in the training data).

### 2.2 Why Self-Attention Solves This

Each of MedScribe's three failure modes maps directly to a limitation that self-attention addresses:

**Long-range dependencies.** In the LSTM, information from token $x_1$ must survive through a chain of $n-1$ hidden state updates to reach token $x_n$. In self-attention, $x_1$ and $x_n$ interact directly through the $QK^T$ dot product -- the path length is $O(1)$.

**Negation handling.** Self-attention computes pairwise scores between all tokens. The token "not" at position $i$ and the diagnosis "pneumonia" at position $j$ will have a direct attention score $q_{\text{not}} \cdot k_{\text{pneumonia}}$, regardless of how far apart they are. A dedicated attention head can learn to fire strongly on negation-diagnosis pairs.

**Multi-label support.** With multi-head attention, different heads can specialize in detecting different diagnostic categories. Head 1 might attend to cardiovascular terms, Head 2 to renal terms, Head 3 to negation patterns. The concatenated output preserves information about all diagnoses simultaneously, rather than compressing everything into a single bottleneck vector.

### 2.3 Model Architecture

The proposed architecture is a **Transformer Encoder + Multi-Label Classification Head**:

1. **Input:** Tokenized clinical note (up to 512 tokens, padded/truncated)
2. **Token Embedding:** Learned embedding layer ( $d_{\text{model}} = 256$ )
3. **Positional Encoding:** Sinusoidal positional encoding
4. **Transformer Encoder:** 4 layers, 8 heads, $d_{ff} = 1024$
5. **Pooling:** Mean pooling over the sequence dimension
6. **Classification Head:** Linear projection to 1,200 output logits
7. **Output:** Sigmoid activation for multi-label prediction (threshold = 0.5)

**Loss function:** Binary Cross-Entropy with Logits (BCE), averaged across all labels.

**Key hyperparameters:**

| Parameter | Value | Rationale |
|---|---|---|
| $d_{\text{model}}$ | 256 | Balances expressiveness and inference speed |
| Number of heads | 8 | $d_k = 256/8 = 32$ per head |
| Number of layers | 4 | Sufficient depth for clinical text; more layers risk overfitting on 2.3M notes |
| $d_{ff}$ | 1024 | Standard 4x expansion |
| Dropout | 0.15 | Moderate regularization for medical text |
| Learning rate | 3e-4 | With linear warmup over 2,000 steps |

---

## 3. Implementation Notebook Structure

The implementation notebook walks through the complete pipeline from data preparation to deployment-ready model.

### 3.1 Environment Setup and Data Loading

```
# Install dependencies
!pip install torch torchtext scikit-learn matplotlib seaborn tqdm -q

import torch
import torch.nn as nn
import torch.nn.functional as F
import math
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader, TensorDataset
from sklearn.metrics import f1_score, precision_recall_curve
from tqdm import tqdm

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

We use a synthetic clinical notes dataset that mimics the statistical properties of real ICD-10 coding data:

```
# TODO: Load and preprocess clinical notes dataset
# - Tokenize notes using a medical vocabulary
# - Pad/truncate to max_length=512
# - Create multi-hot label vectors for ICD-10 codes
# - Split into train/val/test (80/10/10)
```

## 3.2 Tokenization and Vocabulary

```
# TODO: Build a medical vocabulary from the training corpus
# - Minimum frequency threshold: 5 occurrences
# - Special tokens: <PAD>=0, <UNK>=1, <CLS>=2
# - Target vocabulary size: ~15,000 tokens
```

## 3.3 Self-Attention Module Implementation

This is the core component. We implement scaled dot-product attention and multi-head attention from scratch, exactly as derived in the article.

```
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout=0.1):
        super().__init__()
        assert d_model % num_heads == 0
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        B, T, C = x.size()

        # Project and reshape for multi-head
        Q = self.W_q(x).view(B, T, self.num_heads, self.d_k).transpose(1, 2)
        K = self.W_k(x).view(B, T, self.num_heads, self.d_k).transpose(1, 2)
        V = self.W_v(x).view(B, T, self.num_heads, self.d_k).transpose(1, 2)

        # Scaled dot-product attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        attn_weights = self.dropout(F.softmax(scores, dim=-1))
        context = torch.matmul(attn_weights, V)

        # Concatenate and project
        context = context.transpose(1, 2).contiguous().view(B, T, self.d_model)
        return self.W_o(context), attn_weights
```

```
# TODO: Verify the attention module with a test input
# - Create random input of shape (2, 50, 256)
# - Pass through the module
# - Assert output shape is (2, 50, 256)
# - Assert attention weights shape is (2, 8, 50, 50)
# - Assert attention weights sum to 1 along the last dimension
```

## 3.4 Transformer Encoder Block

```
class TransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
```

```
        super().__init__()
        self.attention = MultiHeadSelfAttention(d_model, num_heads, dropout)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model)
        )
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        attn_out, attn_weights = self.attention(x, mask)
        x = self.norm1(x + self.dropout1(attn_out))
        ffn_out = self.ffn(x)
        x = self.norm2(x + self.dropout2(ffn_out))
        return x, attn_weights
```

## 3.5 Full Clinical Note Classifier

```
class ClinicalNoteClassifier(nn.Module):
    def __init__(self, vocab_size, d_model, num_heads, d_ff,
                 num_layers, num_classes, max_len=512, dropout=0.15):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=0)
        self.pos_encoding = SinusoidalPositionalEncoding(d_model, max_len)
        self.blocks = nn.ModuleList([
            TransformerBlock(d_model, num_heads, d_ff, dropout)
            for _ in range(num_layers)
        ])
        self.classifier = nn.Linear(d_model, num_classes)
        self.dropout = nn.Dropout(dropout)
        self.d_model = d_model

    def forward(self, x, mask=None):
        # Embedding + PE
        x = self.embedding(x) * math.sqrt(self.d_model)
        x = self.pos_encoding(x)
        x = self.dropout(x)

        # Transformer blocks
        all_attn = []
        for block in self.blocks:
            x, attn = block(x, mask)
            all_attn.append(attn)

        # Mean pooling (ignoring padding)
        if mask is not None:
            mask_expanded = mask[:, 0, 0, :].unsqueeze(-1).float()
            x = (x * mask_expanded).sum(dim=1) / mask_expanded.sum(dim=1)
        else:
            x = x.mean(dim=1)

        logits = self.classifier(x)
        return logits, all_attn
```

```
# TODO: Instantiate the model with the specified hyperparameters
# - vocab_size=15000, d_model=256, num_heads=8
# - d_ff=1024, num_layers=4, num_classes=1200
# - Print total parameter count
```

## 3.6 Training Loop with Multi-Label BCE Loss

```
# TODO: Implement the training loop
# - Use BCEWithLogitsLoss for multi-label classification
# - Adam optimizer with lr=3e-4
# - Linear warmup for 2000 steps
# - Train for 15 epochs
```

```
# - Track loss and micro-F1 score per epoch
# - Save best model checkpoint based on validation F1
```

### 3.7 Evaluation and Metrics

```
# TODO: Evaluate the trained model
# - Compute per-class F1 scores
# - Compute micro and macro F1
# - Generate precision-recall curves for top-5 most common codes
# - Compare against the LSTM baseline metrics
```

### 3.8 Attention Visualization for Clinical Interpretability

This section is critical for clinical deployment -- physicians need to understand *why* the model assigned a particular code.

```
# TODO: Visualize attention patterns on sample clinical notes
# - Show which tokens the model attends to when predicting each code
# - Highlight negation-diagnosis attention patterns
# - Compare attention patterns across different heads
# - Demonstrate that specific heads specialize in specific roles
```

### 3.9 Ablation Studies

```
# TODO: Run ablation experiments
# - Remove positional encoding: measure impact on long-range accuracy
# - Reduce number of heads from 8 to 1: measure impact on multi-label recall
# - Remove scaling (sqrt(d_k)): measure impact on training stability
# - Remove residual connections: measure impact on convergence
```

---

## 4. Production and System Design Extension

### 4.1 Deployment Architecture

The production system processes clinical notes in near-real-time as physicians complete their documentation:

```
Clinical Note (EHR) -> Tokenizer Service -> Attention Model (GPU)
     -> Code Predictions + Confidence Scores + Attention Maps
          -> Coding Dashboard (Coder Review)
               -> Final Codes -> Billing System
```

**Infrastructure decisions:**

- **Model serving:** NVIDIA Triton Inference Server with TensorRT optimization. The attention model is converted to ONNX format for cross-platform compatibility.
- **Batching:** Dynamic batching with max batch size 32 and max wait time 50ms. Clinical notes arrive asynchronously, so batching amortizes GPU overhead.
- **Caching:** Frequently seen phrases (e.g., "history of hypertension") get pre-computed attention patterns cached in Redis, reducing inference time by ~30%.

## 4.2 Scaling Considerations

| Scale Factor | Current | 12-Month Target | Approach |
|---|---|---|---|
| Notes/day | 15,000 | 80,000 | Horizontal scaling with 4 GPU instances |
| Vocabulary | 15,000 | 30,000 | Subword tokenization (BPE) |
| Label space | 1,200 codes | 5,000 codes | Hierarchical classification with ICD-10 tree structure |
| Sequence length | 512 tokens | 2,048 tokens | Efficient attention (Linformer or Flash Attention) |

## 4.3 Regulatory and Ethical Considerations

Clinical NLP systems operate under strict regulatory requirements:

- **HIPAA compliance:** All patient data is de-identified before processing. The model never stores raw clinical text.
- **FDA classification:** The system is classified as a Clinical Decision Support (CDS) tool, not a diagnostic device. All predictions require human review before submission.
- **Bias monitoring:** Monthly audits check for disparities in coding accuracy across patient demographics (age, sex, race, primary language). Any disparity exceeding 3 percentage points triggers a model retraining with rebalanced data.
- **Explainability:** The attention visualization module provides token-level explanations for every code prediction, satisfying the "right to explanation" requirements of hospital accreditation bodies.

## 4.4 Expected Business Impact

| Metric | Before | After | Annual Savings |
|---|---|---|---|
| Coding accuracy | 78% | 88% | \$1.8M in reduced claim denials |
| Coder productivity | 22 notes/hour | 35 notes/hour | \$480K in labor savings |
| Negation errors | 8% FPR | 2% FPR | \$120K in avoided penalties |
| Time to code (avg) | 4.2 min | 2.1 min | 12,000 coder-hours/year |
| **Total annual impact** | | | **\$2.4M** |

The self-attention architecture delivers measurable improvements across every failure mode that motivated the migration from the LSTM baseline. The attention mechanism's ability to directly model long-range dependencies, negation patterns, and multi-label relationships -- without the sequential bottleneck -- makes it the right architectural choice for clinical documentation intelligence.