main_pipeline

March 20, 2025

1 Shorthand to text transformer architecture

1.1 Setup and Imports

[33]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
```

1.2 Compile smaller data selection (random)

[34]:
```
# import os
# import shutil
# import random
# from pathlib import Path

# def create_mini_dataset(source_dir='data', target_dir='data-Mini',␣
↪percentage=0.01):
#     # Create target directory if it doesn't exist
#     Path(target_dir).mkdir(parents=True, exist_ok=True)

#     # Get all image files
#     image_files = [f for f in os.listdir(source_dir)
#                    if f.lower().endswith(('.png', '.jpg', '.jpeg'))]

#     # Calculate number of files to select
#     num_files = len(image_files)
#     num_files_to_select = max(1, int(num_files * percentage))

#     # Randomly select files
#     selected_files = random.sample(image_files, num_files_to_select)

#     # Copy selected files to new directory
#     for file_name in selected_files:
#         source_path = os.path.join(source_dir, file_name)
#         target_path = os.path.join(target_dir, file_name)
```

```
#         shutil.copy2(source_path, target_path)

#     print(f"Original dataset size: {num_files} images")
#     print(f"Mini dataset size: {num_files_to_select} images")
#     print(f"Mini dataset created in: {target_dir}")

# # Create the mini dataset
# # Example with different parameters
# create_mini_dataset(
#     source_dir='data',
#     target_dir='data_mini',
#     percentage=0.01  # 5% instead of 1%
# )
```

1.3 Data Loading

[35]:
```
import torchvision
from torchvision import transforms
from PIL import Image
from torch.utils.data import Dataset, DataLoader
import os

class KeepAspectRatioPad(object):
    """
    Resize the image to fit within the target size while preserving aspect␣
↪ratio,
    then pad the result to the target size.
    """
    def __init__(self, target_size=(64, 64), fill=255):
        self.target_size = target_size
        self.fill = fill  # Color to use for padding

    def __call__(self, img):
        # Get original dimensions
        w_orig, h_orig = img.size
        w_target, h_target = self.target_size

        # Determine scale factor to maintain aspect ratio
        aspect_ratio = w_orig / h_orig

        if aspect_ratio > 1:
            # Image is wider than tall
            new_w = min(w_orig, w_target)
            new_h = int(new_w / aspect_ratio)
        else:
            # Image is taller than wide (or square)
            new_h = min(h_orig, h_target)
```

```
            new_w = int(new_h * aspect_ratio)

        # Resize maintaining aspect ratio
        resized_img = img.resize((new_w, new_h), Image.LANCZOS)

        # Create new image with target size and paste resized image
        new_img = Image.new(img.mode, self.target_size, self.fill)

        # Calculate position for pasting (center)
        paste_x = (w_target - new_w) // 2
        paste_y = (h_target - new_h) // 2

        # Paste resized image onto padded background
        new_img.paste(resized_img, (paste_x, paste_y))

        return new_img

transform = transforms.Compose([
    transforms.Grayscale(),  # Convert to single channel if not already
    KeepAspectRatioPad(target_size=(64, 64), fill=0),  # Preserve aspect ratio␣
 ↪and pad
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485], std=[0.229])  # For grayscale
])
```

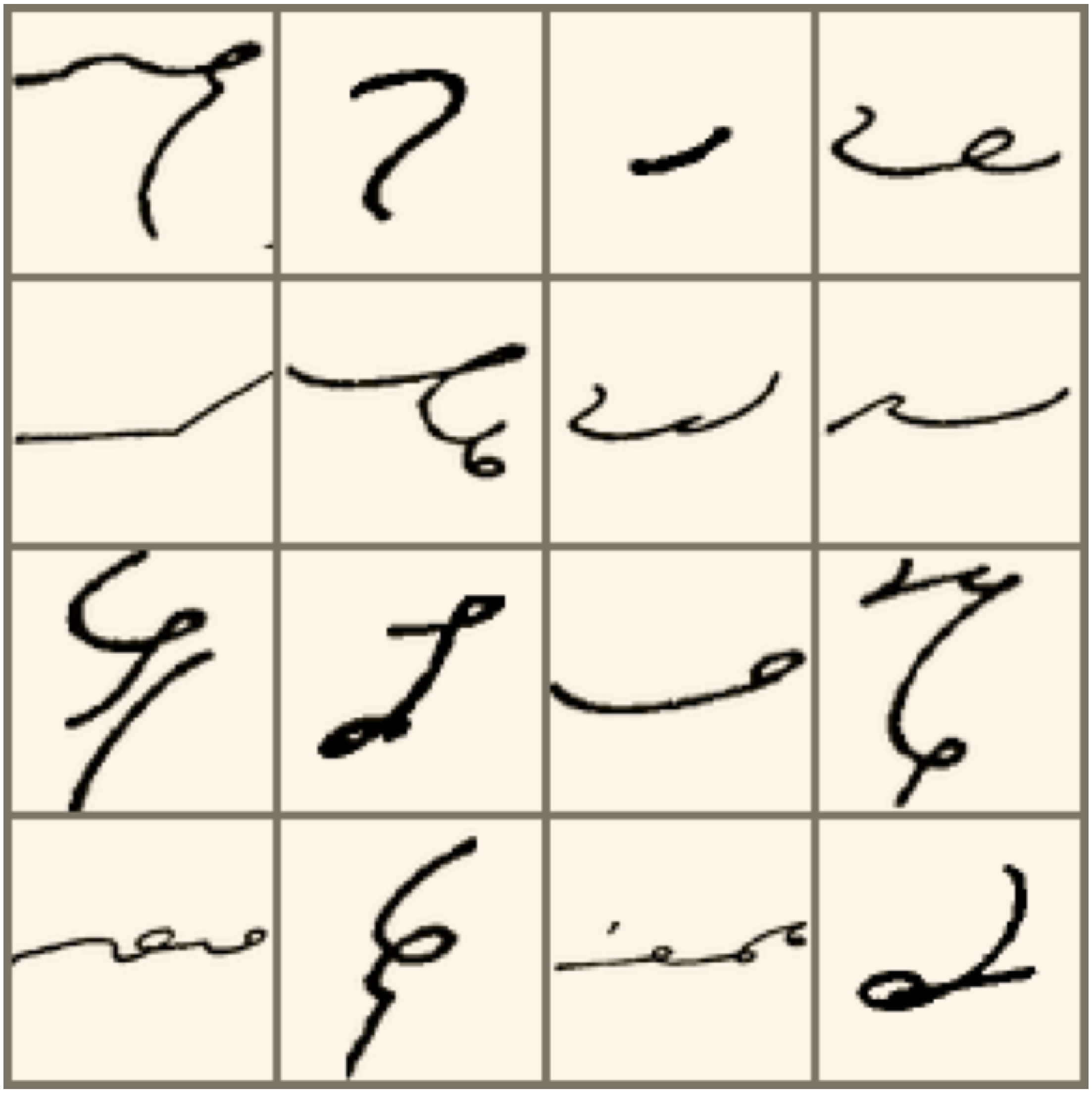
[36]:
```
class ShorthandDataset(Dataset):
    def __init__(self, data_dir, transform=None):
        self.data_dir = data_dir
        self.transform = transform or transforms.Compose([
            transforms.Grayscale(),
            KeepAspectRatioPad(target_size=(64, 64), fill=255),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485], std=[0.229])
        ])
        self.image_files = [f for f in os.listdir(data_dir)
                           if f.endswith(('.png', '.jpg', '.jpeg'))]
        self.labels = [os.path.splitext(f)[0] for f in self.image_files]

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, idx):
        img_path = os.path.join(self.data_dir, self.image_files[idx])
        image = Image.open(img_path).convert('RGB')

        if self.transform:
            image = self.transform(image)
```

```
        return image, self.labels[idx]
```

[37]:
```
# Define transforms
transform = transforms.Compose([
            transforms.Grayscale(),
            KeepAspectRatioPad(target_size=(64, 64), fill=255),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485], std=[0.229])
        ])

# Now create the dataset
dataset = ShorthandDataset('data', transform=transform)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=0)
```

[38]:
```
# Create a function to visualize the results
def visualize_undistorted_batch(dataloader):
    """
    Visualize a batch of images to verify they're not stretched
    """
    images, labels = next(iter(dataloader))

    # Create a grid of images
    grid = torchvision.utils.make_grid(images[:16], nrow=4)

    # Unnormalize
    if grid.shape[0] == 1:  # Grayscale
        mean = torch.tensor([0.485]).view(1, 1, 1)
        std = torch.tensor([0.229]).view(1, 1, 1)
    else:  # RGB
        mean = torch.tensor([0.485, 0.456, 0.406]).view(3, 1, 1)
        std = torch.tensor([0.229, 0.224, 0.225]).view(3, 1, 1)

    grid = grid * std + mean
    grid = torch.clamp(grid, 0, 1)

    # Display
    plt.figure(figsize=(10, 10))
    plt.imshow(grid.permute(1, 2, 0).numpy(), cmap='gray')
    plt.axis('off')
    plt.title("Undistorted Shorthand Images")
    plt.show()

    print(f"Batch shape: {images.shape}")
    print(f"Sample labels: {labels[:5]}")
visualize_undistorted_batch(dataloader)
```

Undistorted Shorthand Images



```
Batch shape: torch.Size([32, 1, 64, 64])
Sample labels: ('increasible', 'company', 'there', 'flare', 'mandate')
```

1.4 Attention Layer

[39]:
```python
class AttentionLayer(nn.Module):
    def __init__(self, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.attention = nn.Linear(hidden_size, 1)

    def forward(self, x):
```

```
        # x shape: [batch, seq_len, hidden]
        attention_scores = self.attention(x)  # [batch, seq_len, 1]
        attention_weights = F.softmax(attention_scores, dim=1)
        context = torch.sum(x * attention_weights, dim=1)  # [batch, hidden]
        return context, attention_weights
```

1.5 Main Model

[40]:
```
class ShorthandModel(nn.Module):
    def __init__(self, num_phonetic_units, phonetic_units, dropout_rate=0.3):
        super(ShorthandModel, self).__init__()
        self.phonetic_units = phonetic_units
        self.blank_idx = num_phonetic_units
        self.lstm_hidden_size = 512


        # Define convolutional layers
        self.conv1 = nn.Conv2d(1, 32, 3)
        self.bn1 = nn.BatchNorm2d(32)
        self.dropout1 = nn.Dropout2d(dropout_rate)

        self.conv2 = nn.Conv2d(32, 64, 3)
        self.bn2 = nn.BatchNorm2d(64)
        self.dropout2 = nn.Dropout2d(dropout_rate)

        self.conv3 = nn.Conv2d(64, 128, 3)
        self.bn3 = nn.BatchNorm2d(128)
        self.dropout3 = nn.Dropout2d(dropout_rate)


        # Check size after convolutions
        self.calc_conv_output_size()

        # Dense layer to prepare for LSTM
        self.fc_prep = nn.Linear(self.conv_output_size, 512)
        self.dropout_fc = nn.Dropout(dropout_rate)


        # LSTM layers
        self.lstm = nn.LSTM(
            input_size=512,          # Size of each time step input
            hidden_size=512,         # Size of LSTM hidden state
            num_layers=3,            # Number of LSTM layers
            bidirectional=True,      # Use bidirectional LSTM
            batch_first=True,         # Batch dimension first
            dropout=dropout_rate if dropout_rate > 0 else 0
```

```
        )

        self.attention = AttentionLayer(self.lstm_hidden_size * 2)  # *2 for bidirectional

        # Output layer - num_phonetic_units + 1 for blank token (CTC requirement)
        self.output = nn.Linear(self.lstm_hidden_size * 2, num_phonetic_units + 1)  # *2 for bidirectional

        # Add training-specific attributes
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.criterion = nn.CTCLoss(blank=num_phonetic_units)  # blank token added

        self._initialize_weights()

        # Add dictionary to map indices to phonetic units
        self.idx_to_phonetic = {i: unit for i, unit in enumerate(phonetic_units)}
        self.idx_to_phonetic[num_phonetic_units] = '<blank>'  # Add blank token

    def calc_conv_output_size(self):
        # Helper function to calculate conv output size
        # 1=Batch size, 1:Num of color channels, (64,64):Image size
        x = torch.randn(1, 1, 64, 64)
        # Apply convolutions with pooling
        x = F.max_pool2d(F.relu(self.bn1(self.conv1(x))), 2)
        x = F.max_pool2d(F.relu(self.bn2(self.conv2(x))), 2)
        x = F.max_pool2d(F.relu(self.bn3(self.conv3(x))), 2)
        # Get dimensions after convolutions
        self.conv_output_shape = x.shape
        _, C, H, W = x.shape
        # Total flattened size
        self.conv_output_size = C * H * W
        # Print sizes for debugging
        print(f"Conv output shape: {x.shape}")
        print(f"Conv output size: {self.conv_output_size}")


    def forward(self, x):
        batch_size = x.size(0)

        # Convolutional layers with regularization
        x = self.conv1(x)
```

```
        x = self.bn1(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)

        x = self.conv2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout2(x)

        x = self.conv3(x)
        x = self.bn3(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout3(x)

        # Apply the same dynamic FC layer but with safety checks
        _, C, H, W = x.size()

        # For debugging
        if W == 0 or H == 0:
            print(f"Error: Invalid dimensions: C={C}, H={H}, W={W}")
            return torch.zeros(batch_size, 1, len(self.phonetic_units) + 1,␣
↪device=self.device)

        x = x.permute(0, 3, 1, 2)  # [batch, width, channels, height]
        batch_size, seq_len, channels, height = x.size()

        # Reshape with safety check
        try:
            x = x.reshape(batch_size * seq_len, channels * height)
        except RuntimeError as e:
            print(f"Reshape error: {e}")
            print(f"Sizes: batch={batch_size}, seq={seq_len}, C={channels},␣
↪H={height}")
            # Return zeros as fallback to avoid crash
            return torch.zeros(batch_size, seq_len, len(self.phonetic_units) +␣
↪1, device=self.device)

        # Dynamically recreate fc_prep if needed
        required_input_size = channels * height
        if not hasattr(self, 'actual_input_size') or self.actual_input_size !=␣
↪required_input_size:
            self.actual_input_size = required_input_size
            self.fc_prep = nn.Linear(required_input_size, 512).to(self.device)
```

```
            print(f"Recreated fc_prep layer with input size:␣
↪{required_input_size}")

        # Apply FC with NaN check
        x = self.fc_prep(x)
        x = F.relu(x)
        if torch.isnan(x).any():
            print("Warning: NaN values after FC layer")
            x = torch.nan_to_num(x, nan=0.0)

        x = self.dropout_fc(x)
        x = x.view(batch_size, seq_len, 512)

        # LSTM with safety
        try:
            lstm_out, _ = self.lstm(x)
        except RuntimeError as e:
            print(f"LSTM error: {e}")
            return torch.zeros(batch_size, seq_len, len(self.phonetic_units) +␣
↪1, device=self.device)

        # Apply attention to get global context
        context, attention_weights = self.attention(lstm_out)

        # Apply output layer directly to LSTM output
        # Using attention as a separate feature extractor rather than modifying␣
↪sequence
        output = self.output(lstm_out)

        # Return log_probs directly to avoid NaN later
        return F.log_softmax(output, dim=2)

    # def decode_greedy(self, output, lengths=None):
    #     # Get the highest probability unit for each sample
    #     max_probs, max_indices = torch.max(output, dim=1)  # dim=1 for␣
↪phonetic units dimension
    #     batch_size = max_indices.size(0)

    #     results = []
    #     for b in range(batch_size):
    #         idx = max_indices[b].item()
    #         # Convert index to phonetic unit if valid
    #         if idx < len(self.phonetic_units):
    #             results.append(self.phonetic_units[idx])
    #         else:
    #             results.append('')  # Empty string for invalid index
```

```python
    #     return results

    # Improved CTC decoder that properly handles blank tokens
    def decode_ctc(self, log_probs):
        """
        Properly decodes CTC output log probabilities into phonetic sequences
        """
        batch_size, seq_len, num_classes = log_probs.shape

        # Convert to probabilities
        probs = torch.exp(log_probs)

        # Get the most likely class at each timestep
        max_probs, max_indices = torch.max(probs, dim=2)  # [batch_size, seq_len]

        # Process each batch
        results = []
        for b in range(batch_size):
            # Extract indices and their probabilities
            indices = max_indices[b].cpu().numpy()
            prob_values = max_probs[b].cpu().numpy()

            # For debugging - show all tokens for a few examples
            if b == 0:
                print("\nRaw prediction sequence (first 10 tokens):")
                for i in range(min(10, seq_len)):
                    token_idx = indices[i]
                    token = self.phonetic_units[token_idx] if token_idx < len(self.phonetic_units) else '<blank>'
                    print(f"Token {i}: {token} (prob: {prob_values[i]:.4f})")

            # Perform CTC collapsing: remove duplicates and blanks
            collapsed = []
            for i, idx in enumerate(indices):
                # Skip blank tokens
                if idx == self.blank_idx:
                    continue

                # Add token if it's not a duplicate of the previous non-blank token
                if len(collapsed) == 0 or idx != collapsed[-1]:
                    collapsed.append(idx)

            # Convert indices to phonetic units
            # If we end up with an empty sequence, add the second most likely non-blank token
```

```
            if len(collapsed) == 0:
                # Find the most likely non-blank token
                avg_probs = probs[b].mean(dim=0).cpu().numpy()
                # Create a copy and set blank token probability to 0
                non_blank_probs = avg_probs.copy()
                non_blank_probs[self.blank_idx] = 0
                # Get the most likely non-blank token
                best_token = np.argmax(non_blank_probs)
                phonetic_seq = [self.phonetic_units[best_token]] if best_token␣
↪< len(self.phonetic_units) else ['?']
            else:
                phonetic_seq = [self.phonetic_units[idx] if idx < len(self.
↪phonetic_units) else '?'
                                for idx in collapsed]

            results.append(''.join(phonetic_seq))

        return results

    def _initialize_weights(self):
        """Initialize model weights to prevent exploding gradients"""
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',␣
↪nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.xavier_normal_(m.weight)
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.LSTM):
                for name, param in m.named_parameters():
                    if 'weight' in name:
                        nn.init.xavier_uniform_(param)
                    elif 'bias' in name:
                        nn.init.constant_(param, 0)

    def phonetics_to_tensor(self, phonetic_sequences):
        """Convert phonetic sequences to tensor of indices"""
        # Create mapping of phonetic unit to index
        unit_to_idx = {unit: idx for idx, unit in enumerate(self.
↪phonetic_units)}
```

```
        # Convert sequences to index tensors
        tensors = []
        for seq in phonetic_sequences:
            indices = [unit_to_idx.get(unit, 0) for unit in seq]  # Default to␣
↪0 if not found
            tensors.append(torch.tensor(indices, dtype=torch.long))

        # Get sequence lengths for CTC
        seq_lengths = [len(t) for t in tensors]

        # Pad sequences to same length
        max_len = max(seq_lengths)
        padded = torch.full((len(tensors), max_len), len(self.phonetic_units),␣
↪dtype=torch.long)  # pad with blank token
        for i, t in enumerate(tensors):
            padded[i, :len(t)] = t

        return padded.to(self.device), torch.tensor(seq_lengths, dtype=torch.
↪long).to(self.device)

#     def train_batch(self, images, phonetic_sequences, batch_num):
#         """Train on a batch of images"""
#         self.train()
#         self.optimizer.zero_grad()

#         # Prepare data
#         images = images.to(self.device)  # [batch_size, 1, 64, 64]
#         targets = self.phonetics_to_tensor(phonetic_sequences)

#         # Forward pass
#         log_probs = F.log_softmax(self(images), dim=-1)

#         # Calculate loss
#         batch_size = images.size(0)
#         input_lengths = torch.full((batch_size,), log_probs.size(1),␣
↪dtype=torch.long)
#         target_lengths = torch.tensor([len(t) for t in phonetic_sequences],␣
↪dtype=torch.long)

#         loss = self.criterion(log_probs, targets, input_lengths,␣
↪target_lengths)

#         # Backward pass and optimize
#         loss.backward()
#         self.optimizer.step()
```

```
#         self.batchesTrained += 1
#         return loss.item()

#     def infer_batch(self, images):
#         """Recognize phonetic sequences in batch of images"""
#         self.eval()
#         with torch.no_grad():
#             # Prepare data
#             images = images.to(self.device)

#             # Forward pass
#             output = self(images)
#             log_probs = F.log_softmax(output, dim=-1)

#             # Decode
#             decoded_sequences = self.decode_greedy(log_probs)

#         return decoded_sequences


# # def decodeGreggShorthand(self, probabilities, input_lengths):
# #     """Specialized decoder for Gregg shorthand"""
# #     # First pass: Get most likely phonetic units
# #     phonetic_transcription = self.decodeGreedy(probabilities, input_lengths)

# #     # Second pass: Apply phonetic-to-text rules
# #     text_results = []
# #     for transcript in phonetic_transcription:
# #         # Apply contextual rules to convert phonetic to text
# #         # Handle brief forms (common words with special symbols)
# #         # Resolve ambiguities using language model
# #         text = self.phoneticToText(transcript)
# #         text_results.append(text)

# #     return text_results
```

1.6 Translate words to phonetics

[41]:
```
import pickle
import os
import requests
import json
from dotenv import load_dotenv
from collections import Counter

# Load environment variables from .env file
load_dotenv()
```

```
def save_phonetics_to_file(phonetic_dict, output_file='phonetics_output.txt'):
    """Save phonetic dictionary to text file"""
    with open(output_file, 'w') as f:
        f.write("WORD\tPHONETICS\n")
        f.write("-" * 40 + "\n")
        for word, phonetics in phonetic_dict.items():
            f.write(f"{word}\t{''.join(phonetics)}\n")
    print(f"Phonetics saved to {output_file}")

def get_phonetics_from_claude(words):
    """Convert words to phonetics using Claude API"""
    API_URL = "https://api.anthropic.com/v1/messages"
    API_KEY = os.getenv("ANTHROPIC_API_KEY")  # Get API key from environment␣
 ↪variable

    if not API_KEY:
        print("Warning: ANTHROPIC_API_KEY not found. Using fallback phonetics.")
        return {word: [c for c in word.lower()] for word in words}  # Use␣
 ↪characters as fallback

    # Create prompt
    prompt = f"""Convert these English words to IPA (International Phonetic␣
 ↪Alphabet).
Return only the word and its IPA pronunciation, separated by a tab, one per␣
 ↪line. DO NOT INCLUDE  slashes / or stress marks  :

{' '.join(words)}"""

    # Call API
    headers = {
        "Content-Type": "application/json",
        "x-api-key": API_KEY,
        "anthropic-version": "2023-06-01"
    }

    data = {
        "model": "claude-3-7-sonnet-20250219",
        "max_tokens": 50000,
        "messages": [{"role": "user", "content": prompt}]
    }

    try:
        response = requests.post(API_URL, headers=headers, json=data)
        response.raise_for_status()
        result = response.json()
        text = result["content"][0]["text"]
```

```
        # Parse response
        phonetic_dict = {}
        for line in text.strip().split('\n'):
            if '\t' in line:
                word, ipa = line.split('\t')
                word = word.strip()
                # Keep all phonetic characters, not just vowels
                phonetic_dict[word] = list(ipa.strip())

        return phonetic_dict

    except Exception as e:
        print(f"Claude API error: {str(e)}")
        # Fallback to character-level representation
        return {word: list(word.lower()) for word in words}
```

[42]:
```
def train(model, train_loader, val_loader, num_epochs=10):
    # Use a much lower learning rate and L2 regularization
    optimizer = torch.optim.Adam(model.parameters(), lr=0.0001,␣
 ↪weight_decay=1e-6)

    # Gradient clipping to prevent exploding gradients
    clip_value = 0.5

    # Count phonetic unit frequencies across the dataset
    phonetic_counts = Counter()
    num_phonetic_units = len(model.phonetic_units)

    # Load phonetic dictionary
    cache_file = 'phonetic_cache.pkl'
    if os.path.exists(cache_file):
        with open(cache_file, 'rb') as f:
            phonetic_dict = pickle.load(f)
        print(f"Loaded {len(phonetic_dict)} words from phonetic cache")
    else:
        phonetic_dict = {}

    # Store the phonetic dictionary in the model
    model.phonetic_dict = phonetic_dict

    # Count frequency of each phonetic unit in training set
    print("Calculating phonetic unit frequencies...")
    all_phonetic_units = []
    for _, labels in train_loader:
        for label in labels:
            if label in phonetic_dict:
```

```
                phonetic_seq = phonetic_dict.get(label, ['ə'])
                all_phonetic_units.extend(phonetic_seq)

    # Count occurrences
    phonetic_counts = Counter(all_phonetic_units)
    print(f"Phonetic unit counts: {dict(phonetic_counts.most_common(10))}")

    # Create weight tensor based on inverse frequency
    weights = torch.ones(num_phonetic_units + 1, device=model.device)

    # Set blank token weight
    blank_weight = 0.1  # Low weight to discourage blank predictions
    weights[model.blank_idx] = blank_weight

    # Set weights for phonetic units
    for idx, unit in enumerate(model.phonetic_units):
        if unit in phonetic_counts and phonetic_counts[unit] > 0:
            # Inverse frequency weighting with smoothing
            weights[idx] = 1.0 / (phonetic_counts[unit] + 1)

            # Special handling for common units
            if unit == 'ə':
                weights[idx] *= 15.0  # Even stronger penalty for schwa
            elif phonetic_counts[unit] > 200:
                weights[idx] *= 5.0

            # Also boost weights for rare units to encourage their prediction
            elif phonetic_counts[unit] < 50:
                weights[idx] *= 2.0

    # Normalize weights
    weights = weights / weights.mean() * 2.0

    # Print the weights for key tokens
    print(f"Weight for blank token: {weights[model.blank_idx]:.4f}")
    print(f"Weight for 'ə': {weights[model.phonetic_units.index('ə')]:.4f}")
    if 'æ' in model.phonetic_units:
        print(f"Weight for 'æ': {weights[model.phonetic_units.index('æ')]:.4f}")

    # Define an improved weighted CTC loss function
    class WeightedCTCLoss(nn.Module):
        def __init__(self, blank_idx, weights):
            super().__init__()
            self.blank_idx = blank_idx
            self.ctc_loss = nn.CTCLoss(blank=blank_idx, reduction='none', zero_infinity=True)
            self.weights = weights
```

```
    def forward(self, log_probs, targets, input_lengths, target_lengths):
        # Standard CTC loss calculation
        per_sample_loss = self.ctc_loss(log_probs, targets, input_lengths,␣
↪target_lengths)

        # Apply weights based on the target units
        # We'll create a simple scaling factor based on the average weight␣
↪of the target units
        target_weights = torch.ones_like(per_sample_loss, device=model.
↪device)

        # For each sample, calculate average weight of its target units
        for i in range(len(targets)):
            if target_lengths[i] > 0:
                # Get the actual target indices for this sample
                sample_targets = targets[i][:target_lengths[i]]
                # Look up weights for each target index
                sample_weights = torch.tensor([self.weights[idx] for idx in␣
↪sample_targets],
                                            device=model.device)
                # Use average weight as the scaling factor for this␣
↪sample's loss
                target_weights[i] = sample_weights.mean()

        # Return weighted loss
        return (per_sample_loss * target_weights).mean()

# Create the weighted loss with the calculated weights
criterion = WeightedCTCLoss(model.blank_idx, weights)


# Add learning rate scheduler
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=2, verbose=True
)

for epoch in range(num_epochs):
    # Training phase
    model.train()
    total_loss = 0
    valid_batches = 0

    for batch_idx, (images, labels) in enumerate(train_loader):
        # Move data to device
        images = images.to(model.device)
```

```
        # Convert words to phonetic sequences
        phonetic_sequences = []
        for label in labels:
            # Get phonetic sequence from dictionary or default to schwa
            phonetic_seq = phonetic_dict.get(label, ['ə'])
            # Ensure reasonable sequence length
            if len(phonetic_seq) > 20:
                phonetic_seq = phonetic_seq[:20]
            phonetic_sequences.append(phonetic_seq)

        # Forward pass with added stability
        optimizer.zero_grad()

        try:
            # Get model outputs
            outputs = model(images)

            # Apply log softmax with numerical stability
            log_probs = F.log_softmax(outputs, dim=2).clamp(min=-100,␣
↪max=100)

            # Check for NaN/Inf
            if torch.isnan(log_probs).any() or torch.isinf(log_probs).any():
                print(f"Warning: NaN/Inf in log_probs (batch {batch_idx}),␣
↪skipping")
                continue

            # Prepare targets for CTC loss (with sequence length checks)
            targets, target_lengths = model.
↪phonetics_to_tensor(phonetic_sequences)

            # Verify target lengths aren't too long compared to input
            seq_len = log_probs.size(1)
            for i in range(len(target_lengths)):
                if target_lengths[i] > seq_len:
                    target_lengths[i] = seq_len

            # Input lengths are sequence length from model output
            input_lengths = torch.full((images.size(0),), seq_len,
                                        dtype=torch.long).to(model.device)

            # Calculate weighted CTC loss with error handling
            try:
                loss = criterion(log_probs.transpose(0, 1), targets,
                                input_lengths, target_lengths)

                # Check for NaN/Inf
```

```
                    if torch.isnan(loss) or torch.isinf(loss):
                        print(f"Warning: NaN/Inf loss in batch {batch_idx},␣
↪skipping")
                        continue

                    # Backward pass with gradient clipping
                    loss.backward()

                    # Apply gradient clipping
                    torch.nn.utils.clip_grad_norm_(model.parameters(),␣
↪clip_value)

                    # Check gradients for NaN/Inf
                    has_bad_grad = False
                    for name, param in model.named_parameters():
                        if param.grad is not None:
                            if torch.isnan(param.grad).any() or torch.
↪isinf(param.grad).any():
                                print(f"Warning: NaN/Inf gradient in {name},␣
↪skipping update")
                                has_bad_grad = True
                                break

                    if not has_bad_grad:
                        optimizer.step()
                        total_loss += loss.item()
                        valid_batches += 1

                except RuntimeError as e:
                    print(f"CTC Loss error: {e}")
                    continue

            except Exception as e:
                print(f"Error in batch {batch_idx}: {e}")
                continue

            # Print progress with improved formatting to show phonetic␣
↪representations
            if batch_idx % 5 == 0:
                print(f'Epoch {epoch+1}/{num_epochs}, Batch {batch_idx+1}/
↪{len(train_loader)}, '
                      f'Loss: {loss.item():.4f}, Valid batches:␣
↪{valid_batches}')

        # Validation phase
        model.eval()
```

```
        val_loss = 0

        # Inside the validation phase, add this code:
        with torch.no_grad():
            for images, labels in val_loader:
                if len(images) > 0:
                    images = images.to(model.device)

                    # Forward pass
                    outputs = model(images)
                    log_probs = F.log_softmax(outputs, dim=2)

                    # Print distribution of probabilities for first example
                    probs = torch.exp(log_probs[0].mean(dim=0))  # Average over␣
↪time steps
                    top_k = torch.topk(probs, min(10, len(model.
↪phonetic_units)))
                    print("\nProbability distribution for first example:")
                    for i, idx in enumerate(top_k.indices):
                        unit = model.phonetic_units[idx] if idx < len(model.
↪phonetic_units) else '<blank>'
                        print(f"{unit}: {top_k.values[i]:.4f}")

                    # Decode predictions
                    decoded = model.decode_ctc(log_probs[:3])
                    for i in range(min(3, len(decoded))):
                        word = labels[i]
                        phonetic = phonetic_dict.get(word, "")
                        print(f'True: "{word}" {phonetic}, Predicted:␣
↪{decoded[i]}')
                    break

        # Calculate average training loss
        avg_train_loss = total_loss / max(1, valid_batches)

        print(f'Epoch {epoch+1}/{num_epochs}, Train Loss: {avg_train_loss:.4f},␣
↪'
              f'Valid batches: {valid_batches}/{len(train_loader)}')

        # Update learning rate based on validation loss
        scheduler.step(avg_train_loss)

    return model
```

[43]:
```
def test(model, test_loader):
    """
    Test the model and print true target values and predictions
```

```
    """
    model.eval()
    print("\nTest Predictions:")
    print("-" * 50)
    print(f"{'True Target':<30} | {'Predicted Value':<15}")
    print("-" * 50)

    with torch.no_grad():
        for images, labels in test_loader:
            images = images.to(model.device)

            # Forward pass
            outputs = model(images)
            log_probs = F.log_softmax(outputs, dim=2)

            # Decode using CTC
            predicted_sequences = model.decode_ctc(log_probs)

            # Print each prediction
            for true_label, predicted in zip(labels, predicted_sequences):
                print(f"{true_label:<30} | {predicted:<15}")

    return
```

[44]:
```
def predict(model, image_path):
    # Prepare image
    transform = transforms.Compose([
        transforms.Resize((64, 64)),
        transforms.Grayscale(num_output_channels=1),
        transforms.ToTensor(),
        transforms.Normalize([0.5], [0.5])
    ])

    # Open and convert image
    image = Image.open(image_path).convert('RGB')
    # Apply transformations (resize, grayscale, normalize)
    image = transform(image).unsqueeze(0)  # Add batch dimension
    # Move to the appropriate device (CPU/GPU)
    image = image.to(model.device)

    # Get prediction
    model.eval()
    with torch.no_grad():
        # Get model output
        output = model(image)

        # Apply log softmax to get log probabilities
```

```
        log_probs = F.log_softmax(output, dim=2)

        # Use CTC decoder to get the sequence
        predictions = model.decode_ctc(log_probs)

        # Since we're processing a single image, take the first prediction
        prediction = predictions[0] if predictions else ""

    return prediction
```

[45]:
```
# Define device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Define transforms
transform = transforms.Compose([
    transforms.RandomRotation(5),  # Small rotations
    transforms.RandomAffine(0, translate=(0.05, 0.05)),  # Small shifts
    transforms.Resize((64, 64)),
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])
])

# Create dataset
dataset = ShorthandDataset('data', transform=transform)
print(f"Total dataset size: {len(dataset)}")

# Split dataset
train_size = int(0.7 * len(dataset))
val_size = int(0.15 * len(dataset))
test_size = len(dataset) - train_size - val_size

train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(
    dataset,
    [train_size, val_size, test_size],
    generator=torch.Generator().manual_seed(42)
)

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True,␣
↪num_workers=0)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False,␣
↪num_workers=0)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False,␣
↪num_workers=0)
```

```
print(f"Train size: {len(train_dataset)}")
print(f"Validation size: {len(val_dataset)}")
print(f"Test size: {len(test_dataset)}")

# Define phonetic units
phonetic_units = ['æ', 'ə', ' ', 'i:', ' ', 'u:', 'p', 't', 'k', 'b', 'd', 'g',␣
 ↪'m', 'n', 'ŋ',
                  'f', 'v', ' ', 'ð', 's', 'z', ' ', ' ', 'h', 'l', 'r', 'w',␣
 ↪'j']

# Create model
model = ShorthandModel(num_phonetic_units=len(phonetic_units),
                      phonetic_units=phonetic_units)

# Train model
model = train(model, train_loader, val_loader, num_epochs=10)

# Test model
test(model, test_loader)
```

```
Using device: cpu
Total dataset size: 15280
Train size: 10696
Validation size: 2292
Test size: 2292
Conv output shape: torch.Size([1, 128, 6, 6])
Conv output size: 4608
Calculating phonetic unit frequencies…
Phonetic unit counts: {}
Weight for blank token: 0.2064
Weight for 'ə': 2.0641
Weight for 'æ': 2.0641
Recreated fc_prep layer with input size: 768
Epoch 1/10, Batch 1/335, Loss: 35.4830, Valid batches: 1
Epoch 1/10, Batch 6/335, Loss: 29.5895, Valid batches: 6
Epoch 1/10, Batch 11/335, Loss: 15.4728, Valid batches: 11
Epoch 1/10, Batch 16/335, Loss: 2.7325, Valid batches: 16
Epoch 1/10, Batch 21/335, Loss: 1.6352, Valid batches: 21
Epoch 1/10, Batch 26/335, Loss: 1.2390, Valid batches: 26
Epoch 1/10, Batch 31/335, Loss: 0.3870, Valid batches: 31
Epoch 1/10, Batch 36/335, Loss: 0.0660, Valid batches: 36
Epoch 1/10, Batch 41/335, Loss: 0.0092, Valid batches: 41
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[45], line 48
     44 model = ShorthandModel(num_phonetic_units=len(phonetic_units),
```

```
     45                          phonetic_units=phonetic_units)
     47 # Train model
---> 48 model = train(model, train_loader, val_loader, num_epochs=10)
     50 # Test model
     51 test(model, test_loader)

Cell In[42], line 182, in train(model, train_loader, val_loader, num_epochs)
    179                 break
    181 if not has_bad_grad:
--> 182     optimizer.step()
    183     total_loss += loss.item()
    184     valid_batches += 1

File /opt/anaconda3/lib/python3.12/site-packages/torch/optim/optimizer.py:484,␣
↪in Optimizer.profile_hook_step.<locals>.wrapper(*args, **kwargs)
    479         else:
    480             raise RuntimeError(
    481                 f"{func} must return None or a tuple of (new_args,␣
↪new_kwargs), but got {result}."
    482             )
--> 484 out = func(*args, **kwargs)
    485 self._optimizer_step_code()
    487 # call optimizer step post hooks

File /opt/anaconda3/lib/python3.12/site-packages/torch/optim/optimizer.py:89, i␣
↪_use_grad_for_differentiable.<locals>._use_grad(self, *args, **kwargs)
     87     torch.set_grad_enabled(self.defaults["differentiable"])
     88     torch._dynamo.graph_break()
---> 89     ret = func(self, *args, **kwargs)
     90 finally:
     91     torch._dynamo.graph_break()

File /opt/anaconda3/lib/python3.12/site-packages/torch/optim/adam.py:226, in␣
↪Adam.step(self, closure)
    214     beta1, beta2 = group["betas"]
    216     has_complex = self._init_group(
    217         group,
    218         params_with_grad,
   (…)
    223         state_steps,
    224     )
--> 226     adam(
    227         params_with_grad,
    228         grads,
    229         exp_avgs,
    230         exp_avg_sqs,
    231         max_exp_avg_sqs,
    232         state_steps,
```

```
    233         amsgrad=group["amsgrad"],
    234         has_complex=has_complex,
    235         beta1=beta1,
    236         beta2=beta2,
    237         lr=group["lr"],
    238         weight_decay=group["weight_decay"],
    239         eps=group["eps"],
    240         maximize=group["maximize"],
    241         foreach=group["foreach"],
    242         capturable=group["capturable"],
    243         differentiable=group["differentiable"],
    244         fused=group["fused"],
    245         grad_scale=getattr(self, "grad_scale", None),
    246         found_inf=getattr(self, "found_inf", None),
    247     )
    249 return loss

File /opt/anaconda3/lib/python3.12/site-packages/torch/optim/optimizer.py:161,␣
 ↪in _disable_dynamo_if_unsupported.<locals>.wrapper.<locals>.
 ↪maybe_fallback(*args, **kwargs)
    159     return disabled_func(*args, **kwargs)
    160 else:
--> 161     return func(*args, **kwargs)

File /opt/anaconda3/lib/python3.12/site-packages/torch/optim/adam.py:766, in␣
 ↪adam(params, grads, exp_avgs, exp_avg_sqs, max_exp_avg_sqs, state_steps,␣
 ↪foreach, capturable, differentiable, fused, grad_scale, found_inf,␣
 ↪has_complex, amsgrad, beta1, beta2, lr, weight_decay, eps, maximize)
    763 else:
    764     func = _single_tensor_adam
--> 766 func(
    767     params,
    768     grads,
    769     exp_avgs,
    770     exp_avg_sqs,
    771     max_exp_avg_sqs,
    772     state_steps,
    773     amsgrad=amsgrad,
    774     has_complex=has_complex,
    775     beta1=beta1,
    776     beta2=beta2,
    777     lr=lr,
    778     weight_decay=weight_decay,
    779     eps=eps,
    780     maximize=maximize,
    781     capturable=capturable,
    782     differentiable=differentiable,
    783     grad_scale=grad_scale,
```

```
    784     found_inf=found_inf,
    785 )

File /opt/anaconda3/lib/python3.12/site-packages/torch/optim/adam.py:431, in␣
↪_single_tensor_adam(params, grads, exp_avgs, exp_avg_sqs, max_exp_avg_sqs,␣
↪state_steps, grad_scale, found_inf, amsgrad, has_complex, beta1, beta2, lr,␣
↪weight_decay, eps, maximize, capturable, differentiable)
    429         denom = (max_exp_avg_sqs[i].sqrt() / bias_correction2_sqrt).
↪add_(eps)
    430     else:
--> 431         denom = (exp_avg_sq.sqrt() / bias_correction2_sqrt).add_(eps)
    433     param.addcdiv_(exp_avg, denom, value=-step_size)
    435 # Lastly, switch back to complex view

KeyboardInterrupt:
```

1.7 Problem diagnosis and Monitoring

[190]:
```python
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn.functional as F
from sklearn.metrics import confusion_matrix
import seaborn as sns
import time
from collections import defaultdict

class ModelMonitor:
    def __init__(self):
        # Track losses per epoch
        self.train_losses = []
        self.val_losses = []

        # Track per-batch metrics
        self.batch_losses = []
        self.batch_gradients = []
        self.batch_times = []

        # Track prediction metrics
        self.accuracy_history = []
        self.confusion_matrices = []

        # Track layer-specific metrics
        self.layer_activations = defaultdict(list)
        self.weight_norms = defaultdict(list)
        self.gradient_norms = defaultdict(list)
```

```python
        # Track model stability
        self.nan_inf_counts = {'train': 0, 'val': 0}

        # Track phonetic predictions
        self.prediction_stats = defaultdict(int)

    def log_epoch(self, epoch, train_loss, val_loss):
        """Log losses after each epoch"""
        self.train_losses.append(train_loss)
        self.val_losses.append(val_loss)

    def log_batch(self, batch_idx, loss, batch_time, model):
        """Log per-batch metrics"""
        self.batch_losses.append(loss)
        self.batch_times.append(batch_time)

        # Check for NaN/Inf in loss
        if torch.isnan(torch.tensor(loss)) or torch.isinf(torch.tensor(loss)):
            self.nan_inf_counts['train'] += 1

        # Track gradient norms
        total_grad_norm = 0
        for name, param in model.named_parameters():
            if param.grad is not None:
                param_norm = param.grad.data.norm(2).item()
                total_grad_norm += param_norm
                self.gradient_norms[name].append(param_norm)

        self.batch_gradients.append(total_grad_norm)

    def log_layer_stats(self, model):
        """Track layer-specific statistics"""
        for name, param in model.named_parameters():
            # Track weight norms for monitoring weight magnitudes
            self.weight_norms[name].append(param.data.norm(2).item())

    def log_predictions(self, true_labels, predicted_labels, phonetic_units, epoch):
        """Log prediction metrics"""
        # Count occurrences of each predicted phonetic unit
        for pred in predicted_labels:
            for char in pred:
                self.prediction_stats[char] += 1

        # Calculate accuracy (exact match)
        correct = sum(1 for t, p in zip(true_labels, predicted_labels) if t == p)
```

```python
        accuracy = correct / len(true_labels) if len(true_labels) > 0 else 0
        self.accuracy_history.append(accuracy)

        # Create confusion matrix for most common 10 phonetic units (if applicable)
        if len(phonetic_units) > 1:
            # Simplified confusion matrix for first character of each prediction
            y_true = [phonetic_units.index(true[0]) if len(true) > 0 and true[0] in phonetic_units
                      else 0 for true in true_labels]
            y_pred = [phonetic_units.index(pred[0]) if len(pred) > 0 and pred[0] in phonetic_units
                      else 0 for pred in predicted_labels]

            if len(set(y_pred)) > 1:  # Only if we have variation in predictions
                cm = confusion_matrix(y_true, y_pred)
                self.confusion_matrices.append((epoch, cm))

    def plot_metrics(self, save_path=None):
        """Plot all tracked metrics"""
        fig, axes = plt.subplots(3, 2, figsize=(15, 15))

        # Plot 1: Training and validation loss
        axes[0, 0].plot(self.train_losses, label='Train Loss')
        axes[0, 0].plot(self.val_losses, label='Validation Loss')
        axes[0, 0].set_title('Training and Validation Loss')
        axes[0, 0].set_xlabel('Epoch')
        axes[0, 0].set_ylabel('Loss')
        axes[0, 0].legend()

        # Plot 2: Batch loss
        axes[0, 1].plot(self.batch_losses)
        axes[0, 1].set_title('Batch Loss')
        axes[0, 1].set_xlabel('Batch')
        axes[0, 1].set_ylabel('Loss')

        # Plot 3: Gradient norms over time
        axes[1, 0].plot(self.batch_gradients)
        axes[1, 0].set_title('Gradient Norm')
        axes[1, 0].set_xlabel('Batch')
        axes[1, 0].set_ylabel('L2 Norm')

        # Plot 4: Weight norms for key layers
        key_layers = ['conv1.weight', 'conv3.weight', 'fc_prep.weight', 'lstm.weight_hh_l0', 'output.weight']
        for layer in key_layers:
            if layer in self.weight_norms and len(self.weight_norms[layer]) > 0:
```

```
                axes[1, 1].plot(self.weight_norms[layer], label=layer)
        axes[1, 1].set_title('Weight Norms')
        axes[1, 1].set_xlabel('Checkpoint')
        axes[1, 1].set_ylabel('L2 Norm')
        axes[1, 1].legend()

        # Plot 5: Prediction distribution
        if self.prediction_stats:
            labels = sorted(self.prediction_stats.keys())
            values = [self.prediction_stats[k] for k in labels]
            axes[2, 0].bar(labels, values)
            axes[2, 0].set_title('Prediction Distribution')
            axes[2, 0].set_xlabel('Phonetic Unit')
            axes[2, 0].set_ylabel('Count')

        # Plot 6: Accuracy
        axes[2, 1].plot(self.accuracy_history)
        axes[2, 1].set_title('Model Accuracy')
        axes[2, 1].set_xlabel('Epoch')
        axes[2, 1].set_ylabel('Accuracy')

        plt.tight_layout()

        if save_path:
            plt.savefig(save_path)
        plt.show()

        # Plot confusion matrix separately if available
        if self.confusion_matrices:
            latest_epoch, cm = self.confusion_matrices[-1]
            plt.figure(figsize=(10, 8))
            sns.heatmap(cm, annot=True, fmt='d')
            plt.title(f'Confusion Matrix (Epoch {latest_epoch})')
            plt.xlabel('Predicted')
            plt.ylabel('True')
            if save_path:
                plt.savefig(save_path.replace('.png', '_cm.png'))
            plt.show()

    def get_diagnostic_report(self):
        """Generate a diagnostic report with potential issues"""
        issues = []

        # Check for NaN/Inf values
        if self.nan_inf_counts['train'] > 0:
            issues.append(f"WARNING: Found {self.nan_inf_counts['train']}␣
↪batches with NaN/Inf losses in training")
```

```
        if self.nan_inf_counts['val'] > 0:
            issues.append(f"WARNING: Found {self.nan_inf_counts['val']} batches with NaN/Inf losses in validation")

        # Check for exploding gradients
        if any(g > 10.0 for g in self.batch_gradients):
            max_grad = max(self.batch_gradients)
            issues.append(f"WARNING: Possible exploding gradients detected. Max gradient norm: {max_grad:.2f}")

        # Check for vanishing gradients
        if len(self.batch_gradients) > 10 and all(g < 0.01 for g in self.batch_gradients[-10:]):
            issues.append("WARNING: Possible vanishing gradients detected. Recent gradient norms are very small.")

        # Check for identical predictions
        if len(self.prediction_stats) <= 1:
            issues.append("WARNING: Model making same prediction for all inputs. Possible mode collapse.")

        # Accuracy check
        if self.accuracy_history and max(self.accuracy_history) < 0.01:
            issues.append("WARNING: Very low accuracy throughout training. Model may not be learning.")

        # Print weight norm warnings for layers with unusual growth
        for layer, norms in self.weight_norms.items():
            if len(norms) > 2 and norms[-1] > 3 * norms[0]:
                issues.append(f"WARNING: Layer '{layer}' weights grew by {norms[-1]/norms[0]:.1f}x during training")

        return "\n".join(issues) if issues else "No major issues detected."


def apply_monitoring_to_train_function(train_fn, model_monitor):
    """
    Modify the training function to include monitoring
    This provides a template for how to integrate the monitor
    """
    def monitored_train(model, train_loader, val_loader, num_epochs=10):
        """
        Monitored version of the train function
        """
```

```
    optimizer = torch.optim.Adam(model.parameters(), lr=0.0001,␣
↪weight_decay=1e-5)
    clip_value = 0.5  # Lower clip value for gradient clipping

    # For debugging CTC loss specifically
    ctc_loss = torch.nn.CTCLoss(blank=model.blank_idx, reduction='mean',␣
↪zero_infinity=True)

    # Track intermediate values for debugging
    debug_info = {
        'last_conv_output': None,
        'last_lstm_output': None,
        'last_logits': None,
        'last_log_probs': None,
        'last_targets': None,
        'last_target_lengths': None,
        'last_input_lengths': None
    }

    for epoch in range(num_epochs):
        # Training phase
        model.train()
        total_loss = 0
        for batch_idx, (images, labels) in enumerate(train_loader):
            start_time = time.time()
            images = images.to(model.device)

            # Convert words to phonetic sequences
            phonetic_sequences = []
            for label in labels:
                phonetic_seq = model.phonetic_dict.get(label, ['æ'])
                phonetic_sequences.append(phonetic_seq)

            # Forward pass with debug capture
            optimizer.zero_grad()
            try:
                outputs = model(images)
                # Apply log softmax if not already done in forward pass
                log_probs = F.log_softmax(outputs, dim=2)

                # Save last outputs for debugging
                debug_info['last_logits'] = outputs[0].detach().cpu().
↪numpy()
                debug_info['last_log_probs'] = log_probs[0].detach().cpu().
↪numpy()

                # Prepare targets for CTC loss
```

```
                    targets, target_lengths = model.
↪phonetics_to_tensor(phonetic_sequences)
                    debug_info['last_targets'] = targets
                    debug_info['last_target_lengths'] = target_lengths

                    # Input lengths (output sequence length from model)
                    input_lengths = torch.full((images.size(0),), log_probs.
↪size(1),
                                             dtype=torch.long).to(model.device)
                    debug_info['last_input_lengths'] = input_lengths

                    # Calculate loss with detailed error catching
                    try:
                        loss = ctc_loss(log_probs.transpose(0, 1), targets,␣
↪input_lengths, target_lengths)
                    except Exception as e:
                        print(f"CTC Loss Error: {str(e)}")
                        print(f"log_probs shape: {log_probs.shape}")
                        print(f"targets shape: {targets.shape}")
                        print(f"input_lengths: {input_lengths}")
                        print(f"target_lengths: {target_lengths}")
                        raise

                    # Check for NaN/Inf
                    if torch.isnan(loss) or torch.isinf(loss):
                        print(f"Warning: NaN/Inf loss detected in batch␣
↪{batch_idx}")
                        print(f"log_probs min/max: {log_probs.min().item()},␣
↪{log_probs.max().item()}")

                        # Skip this batch if loss is NaN/Inf
                        if torch.isnan(loss):
                            print("Skipping batch due to NaN loss")
                            continue

                    # Backward pass with gradient clipping
                    loss.backward()
                    torch.nn.utils.clip_grad_norm_(model.parameters(),␣
↪clip_value)

                    # Check gradients for NaN/Inf
                    for name, param in model.named_parameters():
                        if param.grad is not None:
                            if torch.isnan(param.grad).any() or torch.
↪isinf(param.grad).any():
                                print(f"NaN/Inf gradient in {name}")
```

```
                    optimizer.step()

                except Exception as e:
                    print(f"Error in training loop: {str(e)}")
                    print(f"Current batch shape: {images.shape}")

                    # Print debug info on error
                    for k, v in debug_info.items():
                        if v is not None:
                            if isinstance(v, torch.Tensor):
                                print(f"{k} shape: {v.shape}")
                            else:
                                print(f"{k}: {v}")
                    raise

                # Log metrics
                batch_time = time.time() - start_time
                model_monitor.log_batch(batch_idx, loss.item(), batch_time,␣
↪model)

                # Log layer stats periodically
                if batch_idx % 5 == 0:
                    model_monitor.log_layer_stats(model)

                total_loss += loss.item()

                if batch_idx % 10 == 0:
                    print(f'Epoch {epoch+1}/{num_epochs}, Batch {batch_idx+1}/
↪{len(train_loader)}, Loss: {loss.item():.4f}, Time: {batch_time:.2f}s')

            # Validation phase
            model.eval()
            val_loss = 0
            all_true_labels = []
            all_predicted_labels = []

            with torch.no_grad():
                for images, labels in val_loader:
                    images = images.to(model.device)

                    # Convert words to phonetic sequences
                    phonetic_sequences = []
                    for label in labels:
                        phonetic_seq = model.phonetic_dict.get(label, ['æ'])
                        phonetic_sequences.append(phonetic_seq)
```

```
                # Forward pass
                outputs = model(images)
                log_probs = F.log_softmax(outputs, dim=2)

                # Prepare targets for CTC loss
                targets, target_lengths = model.phonetics_to_tensor(phonetic_sequences)
                input_lengths = torch.full((images.size(0),), log_probs.size(1),
                                         dtype=torch.long).to(model.device)

                # Calculate loss
                try:
                    loss = ctc_loss(log_probs.transpose(0, 1), targets, input_lengths, target_lengths)
                    val_loss += loss.item()

                    # Check for NaN/Inf
                    if torch.isnan(loss) or torch.isinf(loss):
                        self.nan_inf_counts['val'] += 1
                except Exception as e:
                    print(f"Validation CTC loss error: {str(e)}")

                # Decode predictions for evaluation
                decoded = model.decode_ctc(log_probs)

                # Track predictions
                all_true_labels.extend(labels)
                all_predicted_labels.extend(decoded)

                # Print some examples
                if len(val_loader) > 0 and images.size(0) > 3:
                    for i in range(min(3, len(decoded))):
                        print(f"True: {labels[i]}, Predicted: {decoded[i]}")

        # Log epoch metrics
        avg_train_loss = total_loss / len(train_loader)
        avg_val_loss = val_loss / len(val_loader) if len(val_loader) > 0 else float('inf')

        model_monitor.log_epoch(epoch, avg_train_loss, avg_val_loss)
        model_monitor.log_predictions(all_true_labels, all_predicted_labels, model.phonetic_units, epoch)

        print(f'Epoch {epoch+1}/{num_epochs}, Train Loss: {avg_train_loss:.4f}, Val Loss: {avg_val_loss:.4f}')
```

```
        # Save phonetic data with model
        model.phonetic_dict = model.phonetic_dict

        # Generate diagnostic report
        print("\nDiagnostic Report:")
        print(model_monitor.get_diagnostic_report())

        # Plot metrics
        model_monitor.plot_metrics(save_path="model_metrics.png")

        return model

    return monitored_train


# Example usage:
"""
# Create a monitor
monitor = ModelMonitor()

# Wrap the original train function
monitored_train = apply_monitoring_to_train_function(train, monitor)

# Use the monitored version instead
model = monitored_train(model, train_loader, val_loader, num_epochs=10)

# Get insights
monitor.plot_metrics()
print(monitor.get_diagnostic_report())
"""

# Specific debugging function for CTC loss
def debug_ctc_loss(model, log_probs, targets, input_lengths, target_lengths):
    """Debug function specifically for CTC loss issues"""
    print(f"log_probs shape: {log_probs.shape}")
    print(f"targets shape: {targets.shape}")
    print(f"input_lengths: {input_lengths}")
    print(f"target_lengths: {target_lengths}")

    # Check for invalid values in log_probs
    print(f"log_probs contains NaN: {torch.isnan(log_probs).any()}")
    print(f"log_probs contains Inf: {torch.isinf(log_probs).any()}")
    print(f"log_probs min: {log_probs.min().item()}, max: {log_probs.max().
 ↪item()}")

    # Check input/target length relationship (must have input_len >= target_len)
```

```
    valid_lengths = True
    for i in range(len(input_lengths)):
        if input_lengths[i] < target_lengths[i]:
            print(f"Invalid length at index {i}: input_len={input_lengths[i]},␣
↪target_len={target_lengths[i]}")
            valid_lengths = False

    if valid_lengths:
        print("All length checks passed.")

    # Check target values are valid indices
    invalid_targets = targets >= model.blank_idx + 1
    if invalid_targets.any():
        print(f"Invalid target indices found: {targets[invalid_targets]}")
        print(f"Max valid index is: {model.blank_idx}")
    else:
        print("All target indices are valid.")


# Layer activation capture module
class ActivationCapture(torch.nn.Module):
    """Helper module to capture activations during forward pass"""
    def __init__(self, name):
        super().__init__()
        self.name = name
        self.activations = None

    def forward(self, x):
        self.activations = x.detach().clone()
        return x


def add_activation_hooks(model):
    """Add hooks to capture layer activations"""
    hooks = []
    captures = {}

    # Add hook for conv outputs
    capture = ActivationCapture('conv_output')
    model.bn3.register_forward_hook(lambda m, i, o: capture(o))
    captures['conv_output'] = capture

    # Add hook for LSTM outputs
    capture = ActivationCapture('lstm_output')
    model.lstm.register_forward_hook(lambda m, i, o: capture(o[0]))
    captures['lstm_output'] = capture
```

```python
    # Add hook for final outputs
    capture = ActivationCapture('final_output')
    model.output.register_forward_hook(lambda m, i, o: capture(o))
    captures['final_output'] = capture

    return captures


def analyze_model_weights(model):
    """Analyze model weights for potential issues"""
    issues = []

    for name, param in model.named_parameters():
        # Skip bias terms
        if '.bias' in name:
            continue

        # Get statistics
        weight = param.data
        mean = weight.mean().item()
        std = weight.std().item()
        min_val = weight.min().item()
        max_val = weight.max().item()

        # Check for issues
        if std < 0.01:
            issues.append(f"WARNING: Small weight variance in {name} (std={std:.
↪6f})")

        if abs(mean) > 0.1:
            issues.append(f"WARNING: Large mean in {name} (mean={mean:.6f})")

        if max_val > 2.0 or min_val < -2.0:
            issues.append(f"WARNING: Extreme values in {name} (min={min_val:.
↪2f}, max={max_val:.2f})")

    # Print report
    if issues:
        print("\nWeight Analysis Issues:")
        for issue in issues:
            print(issue)
    else:
        print("\nWeight Analysis: No major issues detected.")


# Fix model initialization
def fix_model_initialization(model):
```

```
    """Apply proper initialization to model weights"""
    for m in model.modules():
        if isinstance(m, torch.nn.Conv2d):
            torch.nn.init.kaiming_normal_(m.weight, mode='fan_out',␣
↪nonlinearity='relu')
            if m.bias is not None:
                torch.nn.init.constant_(m.bias, 0)
        elif isinstance(m, torch.nn.BatchNorm2d):
            torch.nn.init.constant_(m.weight, 1)
            torch.nn.init.constant_(m.bias, 0)
        elif isinstance(m, torch.nn.Linear):
            torch.nn.init.xavier_normal_(m.weight)
            if m.bias is not None:
                torch.nn.init.constant_(m.bias, 0)
        elif isinstance(m, torch.nn.LSTM):
            for name, param in m.named_parameters():
                if 'weight' in name:
                    torch.nn.init.orthogonal_(param)
                elif 'bias' in name:
                    torch.nn.init.constant_(param, 0)

    return model
```

[191]:
```
monitor = ModelMonitor()
monitored_train = apply_monitoring_to_train_function(train, monitor)
```

[ ]:
```
model = monitored_train(model, train_loader, val_loader)
```

```
Epoch 1/10, Batch 1/335, Loss: 26.5934, Time: 0.26s
Epoch 1/10, Batch 11/335, Loss: 8.6102, Time: 0.25s
Epoch 1/10, Batch 21/335, Loss: 1.1975, Time: 0.18s
Epoch 1/10, Batch 31/335, Loss: 0.0263, Time: 0.43s
Epoch 1/10, Batch 41/335, Loss: 0.0041, Time: 0.24s
Epoch 1/10, Batch 51/335, Loss: 0.0024, Time: 0.23s
Epoch 1/10, Batch 61/335, Loss: 0.0034, Time: 0.19s
Epoch 1/10, Batch 71/335, Loss: 0.0028, Time: 0.20s
Epoch 1/10, Batch 81/335, Loss: 0.0021, Time: 0.91s
Epoch 1/10, Batch 91/335, Loss: 0.2360, Time: 0.43s
Epoch 1/10, Batch 101/335, Loss: 0.0058, Time: 0.26s
Epoch 1/10, Batch 111/335, Loss: 0.0035, Time: 0.28s
Epoch 1/10, Batch 121/335, Loss: 0.0024, Time: 0.23s
Epoch 1/10, Batch 131/335, Loss: 0.0024, Time: 0.21s
Epoch 1/10, Batch 141/335, Loss: 0.0013, Time: 0.23s
Epoch 1/10, Batch 151/335, Loss: 0.0014, Time: 0.25s
Epoch 1/10, Batch 161/335, Loss: 0.2229, Time: 0.21s
Epoch 1/10, Batch 171/335, Loss: 0.0024, Time: 0.22s
Epoch 1/10, Batch 181/335, Loss: 0.0024, Time: 0.24s
Epoch 1/10, Batch 191/335, Loss: 0.0021, Time: 0.23s
```

```
Epoch 1/10, Batch 201/335, Loss: 0.0033, Time: 0.23s
Epoch 1/10, Batch 211/335, Loss: 0.0058, Time: 0.22s
Epoch 1/10, Batch 221/335, Loss: 0.0031, Time: 0.71s
Epoch 1/10, Batch 231/335, Loss: 0.0031, Time: 0.27s
Epoch 1/10, Batch 241/335, Loss: 0.1982, Time: 0.24s
Epoch 1/10, Batch 251/335, Loss: 0.0033, Time: 0.23s
Epoch 1/10, Batch 261/335, Loss: 0.0029, Time: 0.21s
Epoch 1/10, Batch 271/335, Loss: 0.0024, Time: 0.42s
Epoch 1/10, Batch 281/335, Loss: 0.2092, Time: 0.32s
Epoch 1/10, Batch 291/335, Loss: 0.1594, Time: 0.44s
Epoch 1/10, Batch 301/335, Loss: 0.0032, Time: 0.24s
Epoch 1/10, Batch 311/335, Loss: 0.0022, Time: 0.23s
Epoch 1/10, Batch 321/335, Loss: 0.0014, Time: 0.22s
Epoch 1/10, Batch 331/335, Loss: 0.0012, Time: 0.22s
True: conscious, Predicted: æ
True: overthrow, Predicted: æ
True: cocoon, Predicted: æ
True: borough, Predicted: æ
True: paltriness, Predicted: æ
True: phlegm, Predicted: æ
True: child, Predicted: æ
True: apostolic, Predicted: æ
True: cracked, Predicted: æ
True: mealiness, Predicted: æ
True: inviolate, Predicted: æ
True: downhill, Predicted: æ
True: organ, Predicted: æ
True: sometimes, Predicted: æ
True: signable, Predicted: æ
True: ingenious, Predicted: æ
True: egress, Predicted: æ
True: pansy, Predicted: æ
True: benefactress, Predicted: æ
True: uniformity, Predicted: æ
True: angelical, Predicted: æ
True: lambrequin, Predicted: æ
True: pistol, Predicted: æ
True: cry, Predicted: æ
True: variegate, Predicted: æ
True: abstracted, Predicted: æ
True: intimate, Predicted: æ
True: conscientious, Predicted: æ
True: cactus, Predicted: æ
True: leash, Predicted: æ
True: turf, Predicted: æ
True: assumable, Predicted: æ
True: consist, Predicted: æ
True: squirrel, Predicted: æ
```

```
True: unwary, Predicted: æ
True: inquire, Predicted: æ
True: middle, Predicted: æ
True: eye, Predicted: æ
True: bloodless, Predicted: æ
True: patricide, Predicted: æ
True: renovate, Predicted: æ
True: rap, Predicted: æ
True: garbage, Predicted: æ
True: inception, Predicted: æ
True: profession, Predicted: æ
True: voucher, Predicted: æ
True: bunchiness, Predicted: æ
True: vindicative, Predicted: æ
True: flatter, Predicted: æ
True: transfusion, Predicted: æ
True: whereby, Predicted: æ
True: twaddle, Predicted: æ
True: like, Predicted: æ
True: elusive, Predicted: æ
True: scholastic, Predicted: æ
True: abstract, Predicted: æ
True: sulphurize, Predicted: æ
True: inharmonious, Predicted: æ
True: torrid, Predicted: æ
True: signification, Predicted: æ
True: know, Predicted: æ
True: breeze, Predicted: æ
True: surveillance, Predicted: æ
True: condition, Predicted: æ
True: republish, Predicted: æ
True: freckle, Predicted: æ
True: tassel, Predicted: æ
True: kite, Predicted: æ
True: grievance, Predicted: æ
True: yarm, Predicted: æ
True: bake, Predicted: æ
True: drill, Predicted: æ
True: penal, Predicted: æ
True: purchasable, Predicted: æ
True: miraculous, Predicted: æ
True: immeasure, Predicted: æ
True: peasant, Predicted: æ
True: shuffling, Predicted: æ
True: breadth, Predicted: æ
True: abrogate, Predicted: æ
True: obstacle, Predicted: æ
True: function, Predicted: æ
```

```
True: piecemeal, Predicted: æ
True: beguile, Predicted: æ
True: rhinestone, Predicted: æ
True: conscionable, Predicted: æ
True: selfconfidence, Predicted: æ
True: refrigeration, Predicted: æ
True: catechetic, Predicted: æ
True: agglomerate, Predicted: æ
True: hasp, Predicted: æ
True: condescend, Predicted: æ
True: indignent, Predicted: æ
True: drunk, Predicted: æ
True: lace, Predicted: æ
True: elocution, Predicted: æ
True: valise, Predicted: æ
True: pine, Predicted: æ
True: imprudent, Predicted: æ
True: sprain, Predicted: æ
True: honk, Predicted: æ
True: straightforward, Predicted: æ
True: traffic, Predicted: æ
True: science, Predicted: æ
True: cog, Predicted: æ
True: abashed, Predicted: æ
True: eczema, Predicted: æ
True: subdued, Predicted: æ
True: spur, Predicted: æ
True: onward, Predicted: æ
True: clink, Predicted: æ
True: infer, Predicted: æ
True: doubtful, Predicted: æ
True: abstain, Predicted: æ
True: odor, Predicted: æ
True: dismissal, Predicted: æ
True: doggerel, Predicted: æ
True: axis, Predicted: æ
True: pastorate, Predicted: æ
True: territory, Predicted: æ
True: pot, Predicted: æ
True: jacosely, Predicted: æ
True: superlative, Predicted: æ
True: statistics, Predicted: æ
True: elemental, Predicted: æ
True: attainment, Predicted: æ
True: attentive, Predicted: æ
True: indigestion, Predicted: æ
True: vincible, Predicted: æ
True: bog, Predicted: æ
```

True: mellowness, Predicted: æ
True: armada, Predicted: æ
True: auditorium, Predicted: æ
True: inaction, Predicted: æ
True: radiant, Predicted: æ
True: chauffeur, Predicted: æ
True: astigmatism, Predicted: æ
True: lubrication, Predicted: æ
True: centrode, Predicted: æ
True: trite, Predicted: æ
True: ford, Predicted: æ
True: constituent, Predicted: æ
True: contemplation, Predicted: æ
True: consent, Predicted: æ
True: arsenal, Predicted: æ
True: generate, Predicted: æ
True: perspective, Predicted: æ
True: constrict, Predicted: æ
True: element, Predicted: æ
True: wretchedness, Predicted: æ
True: concede, Predicted: æ
True: orchard, Predicted: æ
True: obituary, Predicted: æ
True: scribble, Predicted: æ
True: cynosure, Predicted: æ
True: lever, Predicted: æ
True: foible, Predicted: æ
True: chemist, Predicted: æ
True: packt, Predicted: æ
True: cypress, Predicted: æ
True: alertness, Predicted: æ
True: predict, Predicted: æ
True: histology, Predicted: æ
True: ivy, Predicted: æ
True: ready, Predicted: æ
True: learnable, Predicted: æ
True: drawn, Predicted: æ
True: skill, Predicted: æ
True: candidacy, Predicted: æ
True: sleek, Predicted: æ
True: monument, Predicted: æ
True: selfish, Predicted: æ
True: accessiblility, Predicted: æ
True: abutment, Predicted: æ
True: conjure, Predicted: æ
True: aperient, Predicted: æ
True: prostrate, Predicted: æ
True: purification, Predicted: æ

```
True: causation, Predicted: æ
True: permeate, Predicted: æ
True: pinnacle, Predicted: æ
True: reprobate, Predicted: æ
True: fumigate, Predicted: æ
True: denizen, Predicted: æ
True: harshness, Predicted: æ
True: all, Predicted: æ
True: promenade, Predicted: æ
True: pew, Predicted: æ
True: conjugation, Predicted: æ
True: deceit, Predicted: æ
True: haggard, Predicted: æ
True: hoist, Predicted: æ
True: hypnotism, Predicted: æ
True: depopulation, Predicted: æ
True: disinfect, Predicted: æ
True: athelete, Predicted: æ
True: chattel, Predicted: æ
True: gear, Predicted: æ
True: column, Predicted: æ
True: deference, Predicted: æ
True: truncal, Predicted: æ
True: plain, Predicted: æ
True: measurable, Predicted: æ
True: pancreatic, Predicted: æ
True: bellow, Predicted: æ
True: derive, Predicted: æ
True: joyfulness, Predicted: æ
True: nor, Predicted: æ
True: compulsorily, Predicted: æ
True: vixen, Predicted: æ
True: pervert, Predicted: æ
True: shrive, Predicted: æ
True: location, Predicted: æ
True: inconsistency, Predicted: æ
True: mite, Predicted: æ
True: somber, Predicted: æ
Epoch 1/10, Train Loss: 0.7179, Val Loss: 0.0529
Epoch 2/10, Batch 1/335, Loss: 0.0011, Time: 0.24s
Epoch 2/10, Batch 11/335, Loss: 0.0008, Time: 0.27s
Epoch 2/10, Batch 21/335, Loss: 0.0006, Time: 0.33s
Epoch 2/10, Batch 31/335, Loss: 0.0006, Time: 0.20s
Epoch 2/10, Batch 41/335, Loss: 0.0006, Time: 0.21s
Epoch 2/10, Batch 51/335, Loss: 0.0017, Time: 0.22s
Epoch 2/10, Batch 61/335, Loss: 0.0125, Time: 0.24s
Epoch 2/10, Batch 71/335, Loss: 0.0049, Time: 0.22s
Epoch 2/10, Batch 81/335, Loss: 0.2309, Time: 0.21s
```

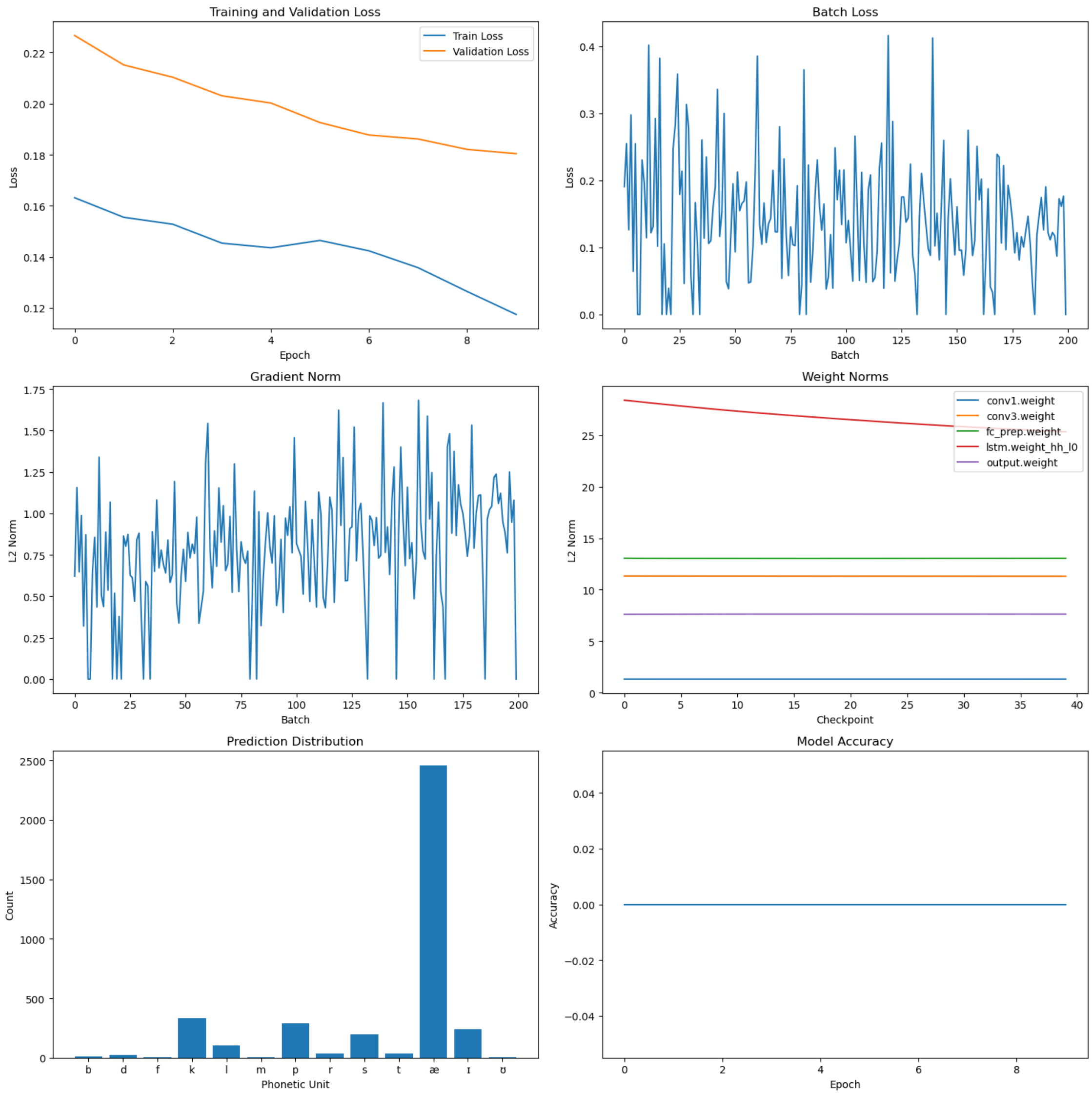
```
Epoch 2/10, Batch 91/335, Loss: 0.0038, Time: 0.27s
Epoch 2/10, Batch 101/335, Loss: 0.0019, Time: 0.25s
Epoch 2/10, Batch 111/335, Loss: 0.0019, Time: 0.23s
Epoch 2/10, Batch 121/335, Loss: 0.0045, Time: 0.20s
Epoch 2/10, Batch 131/335, Loss: 0.0033, Time: 0.23s
Epoch 2/10, Batch 141/335, Loss: 0.0023, Time: 0.23s
Epoch 2/10, Batch 151/335, Loss: 0.0032, Time: 0.24s
Epoch 2/10, Batch 161/335, Loss: 0.0031, Time: 0.30s
Epoch 2/10, Batch 171/335, Loss: 0.0024, Time: 0.23s
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[192], line 1
----> 1 model = monitored_train(model, train_loader, val_loader)

Cell In[190], line 276, in apply_monitoring_to_train_function.<locals>.
 ↪monitored_train(model, train_loader, val_loader, num_epochs)
    273         continue
    275 # Backward pass with gradient clipping
--> 276 loss.backward()
    277 torch.nn.utils.clip_grad_norm_(model.parameters(), clip_value)
    279 # Check gradients for NaN/Inf

File /opt/anaconda3/lib/python3.12/site-packages/torch/_tensor.py:521, in Tensor.
 ↪backward(self, gradient, retain_graph, create_graph, inputs)
    511 if has_torch_function_unary(self):
    512     return handle_torch_function(
    513         Tensor.backward,
    514         (self,),
   (…)
    519         inputs=inputs,
    520     )
--> 521 torch.autograd.backward(
    522     self, gradient, retain_graph, create_graph, inputs=inputs
    523 )

File /opt/anaconda3/lib/python3.12/site-packages/torch/autograd/__init__.py:289
 ↪in backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables
 ↪inputs)
    284     retain_graph = create_graph
    286 # The reason we repeat the same comment below is that
    287 # some Python versions print out the first line of a multi-line functio
    288 # calls in the traceback and some print out the last line
--> 289 _engine_run_backward(
    290     tensors,
    291     grad_tensors_,
    292     retain_graph,
    293     create_graph,
```

```
    294     inputs,
    295     allow_unreachable=True,
    296     accumulate_grad=True,
    297 )

File /opt/anaconda3/lib/python3.12/site-packages/torch/autograd/graph.py:768, in␣
 ↪_engine_run_backward(t_outputs, *args, **kwargs)
    766     unregister_hooks = _register_logging_hooks_on_whole_graph(t_outputs)
    767 try:
--> 768     return Variable._execution_engine.run_backward(  # Calls into the␣
 ↪C++ engine to run the backward pass
    769         t_outputs, *args, **kwargs
    770     )  # Calls into the C++ engine to run the backward pass
    771 finally:
    772     if attach_logging_hooks:

KeyboardInterrupt:
```

[140]:
```
print(monitor.get_diagnostic_report())
monitor.plot_metrics()
```

```
WARNING: Very low accuracy throughout training. Model may not be learning.
```


Training and Validation Loss
Train Loss
Validation Loss
Loss
Epoch
Batch Loss
Loss
Batch
Gradient Norm
L2 Norm
Batch
Weight Norms
conv1.weight
conv3.weight
fc_prep.weight
lstm.weight_hh_l0
output.weight
L2 Norm
Checkpoint
Prediction Distribution
Count
b
d
f
k
l
m
p
r
s
t
æ
ɪ
ʊ
Phonetic Unit
Model Accuracy
Accuracy
Epoch

[ ]:

[ ]: