Neural Speech Decoding: Improving Real-Time Phoneme Prediction from Intracranial Signals

Yanghonghui Chen
Dept. of ECE, UCLA
UID: 706766035
honghui25@g.ucla.edu

Mu Li
Dept. of ECE, UCLA
UID: 306780665
lim29@g.ucla.edu

Kaiwen Zhao
Dept. of ECE, UCLA
UID: 606763706
kaiwenzhao2@g.ucla.edu

Abstract

This project explores the task of decoding text (phonemes) from intracranial neural recordings (area 6v) of a participant with ALS. Starting from a Gated Recurrent Unit (GRU) baseline, we systematically optimize the decoding pipeline through optimizer tuning, architectural enhancements, data augmentation, and loss function engineering. While our optimized LSTM-based model achieved a Phoneme Error Rate (PER) of **19.59%**, it incurred higher computational costs on standard cloud hardware. In a final distillation experiment utilizing high-performance hardware, we demonstrate that a high-capacity GRU, utilizing a specialized non-linear stack and stable optimization, can match this performance (**19.65%**) with significantly reduced training time. This highlights the critical role of optimizer stability and architectural capacity over recurrent complexity in real-time decoding.

1 Introduction

Brain-Computer Interfaces (BCIs) aim to restore communication for individuals with severe speech deficits. This project utilizes the Brain-to-Text Benchmark '24 dataset to decode phonemes from neural activity in the ventral premotor cortex (area 6v)(4). The core challenge is to improve decoding accuracy (minimize PER) while adhering to strict real-time constraints, meaning the model cannot utilize future information (uni-directional processing).

2 Methodology and Experiments

2.1 Experimental Setup and Hardware

To evaluate performance across different compute constraints, experiments were conducted in two distinct environments:

- **Standard Environment (Exp 1-7)**: Conducted on a Google Cloud Platform (GCP) **n1-standard-4** instance (4 vCPUs, 15 GB Memory) equipped with a single **NVIDIA Tesla T4 GPU**. These models were typically trained for 10,000 batches.
- **High-Performance Environment (Exp 8)**: Conducted on a local workstation equipped with an **NVIDIA RTX 5070 Ti (GDDR7)**. This experiment leveraged superior compute to train a larger model for 60,000 batches.

2.2 Baseline Replication

We reproduced the provided baseline model to establish a performance benchmark.

- **Architecture**: 5-layer uni-directional GRU with a hidden dimension of 256.

- **Hyperparameters**: Adam optimizer (LR=0.05, $\epsilon = 0.1$), LinearLR scheduler, White Noise augmentation (SD=0.8).

Result: The baseline achieved a final PER of **23.6%** after 10,000 batches on the T4 GPU, confirming successful replication.

2.3 Experiment 1: Optimization Strategy (Adam $\rightarrow$ AdamW)

Motivation: The baseline uses Adam with weight decay coupled to gradient updates. We hypothesized that **AdamW** (3), which decouples weight decay, would improve generalization. Additionally, we aimed to fix the numerical instability masked by the baseline's unusually high epsilon (0.1).

Implementation:

- Switched to `torch.optim.AdamW` with standard $\epsilon = 1e - 8$.
- Adopted `CosineAnnealingLR` to allow fine-grained convergence at the end of training.
- Reduced LR to 0.002 to prevent gradient explosion caused by the lower epsilon.

Result: The experiment was stopped early at Batch 8500 with a PER of $\sim$40.0%. **Insight**: This demonstrated significant **underfitting**. We discovered the baseline's high learning rate (0.05) was dependent on the high epsilon acting as a "damper." Removing this damper required lowering the LR, but 0.002 proved too conservative for the vanilla GRU architecture.

2.4 Experiment 2: Convergence Acceleration via Stabilized Training

Motivation: Following the instability observed in Exp 1, we hypothesized that the failures were due to optimization instabilities rather than fundamental model flaws. In this experiment, we aimed to stabilize the training pipeline to allow for faster convergence on the T4 GPU.

Implementation:

- **Principled Stabilization**: Introduced Layer Normalization (1) on the GRU outputs and **Gradient Clipping** (max norm = 5.0) to prevent explosions.
- **Label Smoothing**: Implemented an aggressive decay schedule starting at a high initial Learning Rate of 0.03.
- **Large-Batch Training**: Batch size was set to 128.

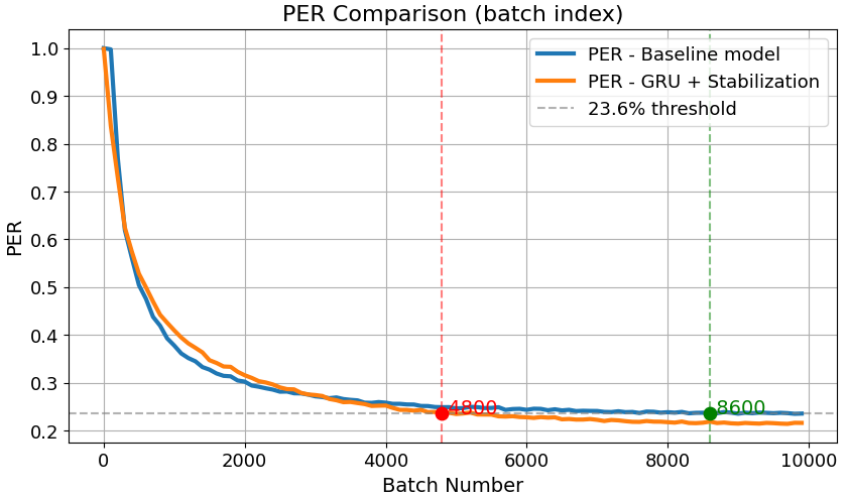

Figure 1: Comparison between PER of baseline model and Experiment 2.

Result: The model converged significantly faster in terms of iterations, reaching baseline parity (23.6%) at only 4,800 batches. However, performance plateaued at **22.1%**.

Insight: This experiment revealed a critical trade-off between *iteration efficiency* and *computational cost*. While the stabilization techniques (Gradient Clipping, LayerNorm) reduced the required batches by $\sim$50%, they introduced significant overhead, increasing the per-batch training time to $\sim$**1.59s** (vs 0.76s Baseline). Consequently, the total wall-clock time to convergence remained comparable to the baseline. This suggests that while we improved sample efficiency, the representational limit of the standard GRU necessitates architectural changes for further PER reduction.

2.5 Experiment 3: Architectural Enhancements (Linderman Lab)

Motivation: To increase model capacity beyond the limits observed in Exp 2. We adopted the "Linderman Lab" approach referenced in the benchmark summary(5).

Implementation:

1. **Post-GRU Stack**: Added a stack of `Linear` $\rightarrow$ `LayerNorm` $\rightarrow$ `Dropout` $\rightarrow$ `GELU` after the GRU output.
2. **Warmup Scheduler**: Implemented a linear warmup (0 to 0.001 over 1000 batches) followed by Cosine Annealing to prevent cold-start instability with AdamW.

Result: PER improved to **21.9%**. **Insight**: **Layer Normalization** successfully stabilized training, preventing the NaN losses seen in Exp 1. The added non-linearity increased expressivity, beating the baseline even with a conservative learning rate (0.001).

2.6 Experiment 4: Data Augmentation

Motivation: To improve robustness and force the network to rely on temporal context rather than memorizing local features. **Implementation**:

- **Time Masking**: Randomly zeroed out contiguous blocks (up to 30 time bins) of neural activity.
- **Increased Noise**: Increased White Noise SD from 0.8 to 1.0.

Result: PER improved slightly to **21.6%**. **Insight**: While the gain was marginal, the model maintained performance despite harder training conditions, indicating improved robustness.

2.7 Experiment 5: Loss Function Engineering (Focal CTC)

Motivation: Standard CTC Loss treats all time steps equally. We hypothesized that the model was mastering easy phonemes while struggling with ambiguous ones. **Implementation**: Replaced standard CTC Loss (2) with **Focal CTC Loss**: $L_{focal} = (1 - p_t)^{\gamma} \cdot L_{CTC}$, with $\gamma = 2.0$. **Result**: PER reached **21.47%**. **Insight**: The loss scale increased significantly due to the weighting, but the gradient direction effectively targeted harder examples. The diminishing returns suggest we are approaching the limits of the GRU architecture.

2.8 Experiment 6: Core Architecture Swap (GRU $\rightarrow$ LSTM)

Motivation: To capture longer-range temporal dependencies that the GRU might miss. Neural speech decoding involves long sequences where early context influences later phonemes. LSTM's separate cell state (c_t) is theoretically better for this task. **Implementation**: Replaced `nn.GRU` with `nn.LSTM`, maintaining the Linderman Post-Stack and Exp 5 hyperparameters.

Result: PER dropped significantly to **19.59%**. **Insight**: This was the most impactful change on the standard hardware. The LSTM structure proved superior for this specific task, likely due to better gradient flow over long sequences. The trade-off is a $\sim$15% increase in training time per batch (0.88s vs 0.76s on T4).

2.9 Experiment 7: Transformer Architecture (Contrastive Study)

Motivation: To evaluate if state-of-the-art Transformer architectures can outperform RNNs in real-time decoding. **Implementation**: Replaced RNN with a 2-layer Transformer Encoder using sinusoidal positional encodings and a **Causal Mask** (to strictly enforce real-time constraints).

Result: PER degraded to **28.0%**. **Insight**:

- **Inductive Bias**: RNNs have a strong inductive bias for sequential data ("state t depends on $t-1$"). Transformers must learn temporal order from scratch via positional encodings, which is difficult with a small dataset (8,800 sentences).
- **Causal Masking**: The strict real-time constraint removes the Transformer's primary advantage (global self-attention). A causal-only Transformer behaves like a less efficient RNN without a recurrent state.

2.10 Experiment 8: Efficiency & Stability (High-Capacity GRU)

Motivation: While the LSTM (Exp 6) achieved the best accuracy, it was computationally expensive on the T4. We hypothesized that with abundant compute, a simpler but larger GRU could match LSTM performance through brute-force capacity.

Implementation:

- **Hardware Shift**: Run on **NVIDIA 5070 Ti** for 60,000 batches.
- **Architecture**: Reverted to GRU but drastically increased capacity (1024 hidden units, 5 layers).
- **Simplified Pipeline**: Disabled Time Masking/Input Dropout; reverted to Adam with high epsilon ($\epsilon = 0.1$) to allow a high initial learning rate (0.02).

Result: The model achieved a PER of **19.65%**, statistically equivalent to the LSTM (19.59%), with a training time of **0.20s/batch** (on 5070 Ti). **Insight**: The raw speedup (4.4x) is primarily due to hardware acceleration. However, achieving parity in accuracy required 6x longer training steps (60k vs 10k), suggesting that the LSTM is more sample-efficient per step, while the High-Capacity GRU favors throughput on high-end hardware.

3 Results and Discussion

Table 1: Summary of Experimental Results. Note: Exp 1-7 run on Tesla T4 (10k batches); Exp 8 run on 5070 Ti (60k batches).

Experiment	Method	Hardware	Val PER	Time/Batch
Baseline	GRU + Adam	T4	23.6%	∼0.76s
Exp 2	GRU + Stabilization	T4	22.1%	∼1.59s
Exp 3	GRU + Linderman Arch	T4	21.9%	∼0.76s
Exp 5	+ Focal Loss	T4	21.47%	∼0.77s
Exp 6	**LSTM + All Above**	**T4**	**19.59%**	**∼0.88s**
Exp 7	Transformer (Causal)	T4	28.0%	∼0.29s
Exp 8	High-Cap GRU (1024u)	5070 Ti	19.65%	∼0.20s*

*Time/Batch reflects newer hardware acceleration (5070 Ti). Exp 2 high time reflects computational overhead of gradient clipping and smoothing.

Through systematic optimization, we improved the neural speech decoding performance from a baseline PER of 23.6% to **19.59%**.

Our findings highlight the nuanced trade-offs between optimization, architecture, and compute resources. Experiment 2 demonstrated that while stabilization techniques like Gradient Clipping allow for faster convergence in terms of epochs, they incur significant computational overhead (∼1.59s/batch), negating wall-clock time gains on limited hardware (T4).

Ultimately, two distinct optimal paths emerged:

1. **Sample Efficiency (Exp 6)**: For standard hardware, the **LSTM** architecture with advanced regularization (Linderman Stack) and Focal Loss yielded the best accuracy (19.59%) with moderate compute costs.

2. **Throughput Scalability (Exp 8)**: With high-end hardware (5070 Ti), a brute-force **High-Capacity GRU** approach matched LSTM performance (19.65%) by leveraging raw throughput (0.20s/batch) to train for 6x more steps.

References

[1] Ba, J. L., Kiros, J. R., and Hinton, G. E. "Layer Normalization." *arXiv preprint arXiv:1607.06450* (2016).

[2] Graves, A., et al. "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks." *Proceedings of the 23rd international conference on Machine learning (ICML)*. (2006).

[3] Loshchilov, I., and Hutter, F. "Decoupled Weight Decay Regularization." *International Conference on Learning Representations (ICLR)*. (2019).

[4] Willett, F. R., et al. "A high-performance speech neuroprosthesis." *Nature* 620, 1031–1036 (2023).

[5] Willett, F. R., et al. "Brain-to-text benchmark '24: Lessons learned." *arXiv preprint arXiv:2412.17227* (2024).

A Appendix: Implementation of Optimal Algorithm

Here we provide the core code snippets for our best-performing model (Experiment 6), which achieved a PER of 19.59%.

A.1 LSTM Decoder Architecture (model.py)

The following class replaces the baseline GRU with an LSTM and implements the "Linderman" post-processing stack (Linear $\rightarrow$ LayerNorm $\rightarrow$ Dropout $\rightarrow$ GELU) to enhance representational capacity.

Listing 1: LSTM Architecture with Linderman Stack

```
class LSTMDecoder(nn.Module):
    def __init__(self, neural_dim, n_classes, hidden_dim, layer_dim,
        nDays=24, dropout=0, device="cuda", ...):
        super(LSTMDecoder, self).__init__()
        # ... (Preprocessing layers omitted for brevity) ...

        # [CORE CHANGE 1] Replaced GRU with LSTM
        self.lstm_decoder = nn.LSTM(
            (neural_dim) * self.kernelLen,
            hidden_dim,
            layer_dim,
            batch_first=True,
            dropout=self.dropout,
            bidirectional=self.bidirectional,
        )

        # [CORE CHANGE 2] Post-LSTM Non-Linear Stack
        # Increases depth and stabilizes gradients with LayerNorm
        lstm_output_dim = hidden_dim * 2 if self.bidirectional else
            hidden_dim
        self.post_lstm_stack = nn.Sequential(
            nn.Linear(lstm_output_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.Dropout(self.dropout),
            nn.GELU()
        )
        self.fc_decoder_out = nn.Linear(hidden_dim, n_classes + 1)

    def forward(self, neuralInput, dayIdx):
        # ... (Preprocessing: Smoothing & Day Calibration) ...

        # [CORE CHANGE 3] LSTM Forward Pass
        # LSTM requires both hidden state (h0) and cell state (c0)
        h0 = torch.zeros(self.layer_dim, transformedNeural.size(0),
            self.hidden_dim, device=self.device)
        c0 = torch.zeros(self.layer_dim, transformedNeural.size(0),
            self.hidden_dim, device=self.device)

        lstm_out, _ = self.lstm_decoder(stridedInputs, (h0, c0))

        # Apply the Linderman Stack
        feat = self.post_lstm_stack(lstm_out)

        # Final classification
        seq_out = self.fc_decoder_out(feat)
        return seq_out
```

A.2 Focal CTC Loss (losses.py)

To address the class imbalance between easy and hard phonemes, we implemented the Focal CTC Loss.

Listing 2: Focal CTC Loss Implementation

```
class FocalCTCLoss(nn.Module):
    def __init__(self, blank=0, gamma=2.0, reduction='mean',
        zero_infinity=True):
        super(FocalCTCLoss, self).__init__()
        self.ctc = nn.CTCLoss(blank=blank, reduction='none',
            zero_infinity=zero_infinity)
        self.gamma = gamma
        self.reduction = reduction

    def forward(self, log_probs, targets, input_lengths,
        target_lengths):
        # Calculate standard CTC loss per sample
        ctc_loss = self.ctc(log_probs, targets, input_lengths,
            target_lengths)

        with torch.no_grad():
            # Calculate weight: (1 - p)^gamma
            # p is approximated by exp(-loss)
            prob = torch.exp(-ctc_loss)
            weights = (1.0 - prob) ** self.gamma

        # Apply focal weights
        loss = weights * ctc_loss

        if self.reduction == 'mean': return loss.mean()
        elif self.reduction == 'sum': return loss.sum()
        return loss
```

A.3 Optimizer and Scheduler Strategy (neural_decoder_trainer.py)

To stabilize the training of the LSTM with AdamW, we implemented a sequential learning rate schedule consisting of a linear warmup followed by cosine annealing.

Listing 3: AdamW and Sequential Scheduler

```
    # 1. Optimizer: AdamW with standard epsilon
    # We use eps=1e-8 (standard) instead of 0.1 (baseline) for better
        precision
    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=args["lrStart"],
        betas=(0.9, 0.999),
        eps=1e-8,
        weight_decay=args["l2_decay"],
    )

    # 2. Sequential Scheduler
    # Phase 1: Linear Warmup (0 -> lrStart) for first 1000 batches
    warmup_scheduler = torch.optim.lr_scheduler.LinearLR(
        optimizer,
        start_factor=0.01,
        end_factor=1.0,
        total_iters=1000
    )

    # Phase 2: Cosine Annealing (lrStart -> 1e-6) for remaining
        batches
    cosine_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
```

```
        optimizer,
        T_max=args["nBatch"] - 1000,
        eta_min=1e-6
    )

    # Combine schedulers
    scheduler = torch.optim.lr_scheduler.SequentialLR(
        optimizer,
        schedulers=[warmup_scheduler, cosine_scheduler],
        milestones=[1000]
    )
```

A.4 Hyperparameter Configuration (train_model.py)

The final hyperparameters used for Experiment 6. Note the increased white noise for robustness and the conservative learning rate for stability.

Listing 4: Final Hyperparameters for Exp 6

```
args = {}
args['outputDir'] = output_path
args['datasetPath'] = data_path

# -- Optimization --
args['lrStart'] = 0.001         # Conservative LR for LSTM stability
args['batchSize'] = 128         # Maximize GPU utilization
args['nBatch'] = 10000          # Total training steps

# -- Architecture --
args['nUnits'] = 256            # Hidden dimension
args['nLayers'] = 5             # Number of LSTM layers
args['bidirectional'] = False   # Real-time constraint

# -- Regularization & Augmentation --
args['dropout'] = 0.2
args['whiteNoiseSD'] = 1.0      # Increased from 0.8 (Baseline)
args['constantOffsetSD'] = 0.2
```