Neural Time-Series Forecasting: Implementation Plan

Factorized Spatiotemporal Encoder with Iterative Refinement

Target Challenge: NSF HDR Neural Forecasting Challenge

Task: Predict 10 future timesteps of neural electrode recordings given 10 observed timesteps

Key Constraint: Cross-session generalization with day-to-day recording drift

Evaluation Metric: Mean Squared Error (MSE) on predicted signals

Version	1.0
Target Audience	Solo Developer
Implementation Language	Python / PyTorch
Estimated Development Time	2-3 weeks

Table of Contents

1. Problem Summary and Constraints

1.1 The Neural Forecasting Task

The NSF HDR Neural Forecasting Challenge presents a multivariate time-series prediction problem using neural electrode recordings from primates performing motor tasks. The core task is to predict future neural activity given a short observation window, with the critical constraint that models must generalize across recording sessions where electrode behavior may drift.

1.2 Data Specifications

Dimension	Specification	Notes
Input Shape	N × 20 × C × 9	Samples × Time × Channels × Features
Observed Window	Timesteps 1-10	Model input
Prediction Target	Timesteps 11-20	Only feature[0] is scored
Monkey A (affi)	239 electrodes	~1,150 training samples
Monkey B (beignet)	87 electrodes	~860 training samples
Features	9 per electrode	Feature[0] = target; [1:8] = frequency bands

1.3 Key Challenges

- **Session Drift:** Test data comes from different recording sessions where electrode responses may have shifted.
- **Limited Data:** With roughly 1,000 samples per monkey, overfitting is a serious risk.
- **Spatial Structure:** The electrodes record from nearby neural populations with correlated activity.
- **Multi-feature Utilization:** The 8 auxiliary frequency-band features provide additional signal.
- **Short Horizon, High Dimensionality:** Predicting 10 steps × 239 channels = 2,390 values per sample.

2. Motivation: Why This Architecture?

2.1 Design Philosophy

Core Design Principles:

1. **Simplicity with strong inductive biases** over architectural complexity
2. **Robustness to distribution shift** over raw in-distribution performance
3. **Efficient use of auxiliary features** (frequency bands are free signal)
4. **Factorized computation** to keep parameter counts manageable

2.2 Why Not Simpler Approaches?

The GRU Baseline Limitations:

- Ignores spatial structure entirely, treating electrode 1 and electrode 239 as equally related.
- The flattened input dimension ($239 \times 9 = 2{,}151$) forces large hidden states, increasing overfitting.
- Cannot share learned temporal dynamics across electrodes.

Why Not a Standard Transformer?

A vanilla Transformer with (timestep, electrode) tokens would have $10 \times 239 = 2{,}390$ tokens. Full self-attention is $O(n^2)$, meaning ~5.7 million attention computations per sample—expensive and prone to overfitting on limited data.

2.3 The Case for Factorized Attention

- **Physical locality:** Neural dynamics are local in time and space.
- **Computational efficiency:** $O(T^2 \cdot C + C^2 \cdot T)$ vs $O((T \cdot C)^2)$—roughly 10x reduction.
- **Implicit regularization:** Limited attention patterns reduce hypothesis space.
- **Interpretability:** Separate temporal and spatial attention weights are easier to debug.

2.4 Why Iterative Refinement?

- Start from a simple initialization (e.g., constant continuation).
- Use the encoder to process concatenated observed + predicted sequence.
- Refine predictions based on learned spatiotemporal dynamics.
- Correct early prediction errors that would otherwise propagate.

2.5 Avoiding Electrode ID Embeddings

Critical Design Decision: This architecture deliberately avoids learned electrode-specific embeddings. While such embeddings would improve training-set performance, they would memorize session-specific characteristics that don't transfer. Instead, electrode identity emerges from the electrode's temporal pattern and correlation structure with other electrodes.

3. Architecture Overview

3.1 The Big Picture

Stage 1 - Feature Encoding: Convert each electrode's 9 features into a compact d-dim representation.

Stage 2 - Spatiotemporal Processing: Let electrodes "talk to each other" across time and space via factorized attention.

Stage 3 - Initial Prediction: Generate a first guess for the next 10 timesteps.

Stage 4 - Iterative Refinement: Feed predictions back through the encoder to correct errors (K=2-3 times).

3.2 Architecture Diagram

```
Input: X [B, 10, C, 9]
       |
       v
+-----------------------------------+
|   Feature Encoder (MLP)           |  9 -> d dimensions (shared)
+-----------------------------------+
       |
       v
+-----------------------------------+
| Sinusoidal Temporal Encoding      |  Add position information
+-----------------------------------+
       |
       v
+-----------------------------------+
| Factorized Block (x L layers)     |
|   - Temporal Self-Attention       |  Each electrode attends to its history
|   - Spatial Self-Attention        |  Electrodes attend to each other
|   - FFN + LayerNorm               |
+-----------------------------------+
       |
       v
+-----------------------------------+
|   Prediction Head                 |  d -> 10 future timesteps
+-----------------------------------+
       |
       v
+-----------------------------------+
| Iterative Refinement (x K)        |  Re-encode and refine
+-----------------------------------+
       |
       v
Output: [B, 10, C]
```

3.3 Data Flow Example (Monkey A)

Stage	Shape	Description
Input	`[1, 10, 239, 9]`	One sample, all features
After Encoder	`[1, 10, 239, 64]`	Compressed to d=64
Temporal Attn	`[239, 10, 64]`	Per-electrode sequences
Spatial Attn	`[10, 239, 64]`	Per-timestep electrode sets
Prediction	`[1, 10, 239]`	Future timesteps

4. Component Specifications

4.1 Feature Encoder

The feature encoder transforms 9 raw features per electrode into a d-dimensional representation. It uses a shared MLP across all electrodes and timesteps.

```
class FeatureEncoder(nn.Module):
    def __init__(self, input_dim=9, hidden_dim=32, output_dim=64):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.GELU(),
            nn.LayerNorm(hidden_dim),
            nn.Linear(hidden_dim, output_dim),
            nn.LayerNorm(output_dim),
        )

    def forward(self, x):
        # x: [B, T, C, F] -> [B, T, C, d]
        return self.encoder(x)
```

4.2 Positional Encoding

```
class SinusoidalPositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=50):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1).float()
        div_term = torch.exp(
            torch.arange(0, d_model, 2).float() *
            (-math.log(10000.0) / d_model)
        )
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe.unsqueeze(0).unsqueeze(2))

    def forward(self, x):
        # x: [B, T, C, d]
        return x + self.pe[:, :x.size(1), :, :]
```

Why sinusoidal? Learned positions can memorize dataset-specific patterns. Sinusoidal encodings transfer more reliably to test data with different temporal statistics.

No spatial positions: We deliberately omit spatial positional encodings, making spatial attention permutation-equivariant for robustness to electrode drift.

4.3 Factorized Spatiotemporal Block

This is the core computational unit. Each block consists of temporal self-attention, spatial self-attention, and a feed-forward network with residual connections.

```
class FactorizedSpatiotemporalBlock(nn.Module):
    def __init__(self, d_model, n_heads, dropout=0.1, ff_mult=4):
        super().__init__()
        # Temporal attention (within each electrode)
        self.temporal_attn = nn.MultiheadAttention(
            d_model, n_heads, dropout=dropout, batch_first=True)
        self.temporal_norm = nn.LayerNorm(d_model)

        # Spatial attention (across electrodes)
        self.spatial_attn = nn.MultiheadAttention(
            d_model, n_heads, dropout=dropout, batch_first=True)
        self.spatial_norm = nn.LayerNorm(d_model)

        # Feed-forward network
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_model * ff_mult),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_model * ff_mult, d_model),
            nn.Dropout(dropout),
        )
        self.ffn_norm = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
```

```
def forward(self, x, temporal_mask=None):
        B, T, C, d = x.shape

        # === Temporal Attention ===
        x_t = x.permute(0, 2, 1, 3).reshape(B * C, T, d)
        attn_out, _ = self.temporal_attn(x_t, x_t, x_t, attn_mask=temporal_mask)
        x_t = self.temporal_norm(x_t + self.dropout(attn_out))
        x = x_t.reshape(B, C, T, d).permute(0, 2, 1, 3)

        # === Spatial Attention ===
        x_s = x.reshape(B * T, C, d)
        attn_out, _ = self.spatial_attn(x_s, x_s, x_s)
        x_s = self.spatial_norm(x_s + self.dropout(attn_out))
        x = x_s.reshape(B, T, C, d)

        # === Feed-Forward ===
        x = self.ffn_norm(x + self.ffn(x))
        return x
```

4.4 Prediction Head

```
class PredictionHead(nn.Module):
    def __init__(self, d_model, n_future=10):
        super().__init__()
        self.head = nn.Sequential(
            nn.Linear(d_model, d_model),
            nn.GELU(),
            nn.Linear(d_model, n_future),
        )

    def forward(self, x):
        # x: [B, T, C, d] - take last timestep
        if x.dim() == 4:
            x = x[:, -1, :, :]  # [B, C, d]
        return self.head(x)  # [B, C, 10]
```

4.5 Complete Model

```
class NeuralForecaster(nn.Module):
    def __init__(self, n_channels, n_features=9, d_model=64,
                 n_heads=4, n_layers=3, n_future=10,
                 n_refinement_iters=2, dropout=0.15):
        super().__init__()
        self.feature_encoder = FeatureEncoder(n_features, 32, d_model)
        self.pos_encoding = SinusoidalPositionalEncoding(d_model)
        self.blocks = nn.ModuleList([
            FactorizedSpatiotemporalBlock(d_model, n_heads, dropout)
            for _ in range(n_layers)
        ])
        self.pred_head = PredictionHead(d_model, n_future)
        self.n_refinement_iters = n_refinement_iters
        self.n_future = n_future

    def encode(self, x):
        z = self.feature_encoder(x)
        z = self.pos_encoding(z)
        for block in self.blocks:
            z = block(z)
        return z

    def forward(self, x, use_refinement=True):
        B, T, C, F = x.shape
        encoded = self.encode(x)
        pred = self.pred_head(encoded).permute(0, 2, 1)  # [B, 10, C]

        if use_refinement and self.n_refinement_iters > 0:
            for _ in range(self.n_refinement_iters):
                pred_features = torch.zeros(B, self.n_future, C, F, device=x.device)
                pred_features[:, :, :, 0] = pred
                full_seq = torch.cat([x, pred_features], dim=1)
                full_encoded = self.encode(full_seq)
                future_encoded = full_encoded[:, T:, :, :]
                pred = self.pred_head(future_encoded.mean(dim=1)).permute(0, 2, 1)
        return pred
```

5. Data Pipeline

5.1 Normalization Strategy

Critical Decision: Use per-sample normalization rather than dataset-global statistics. This is essential for handling session drift, where signal magnitude may change between sessions.

```
class PerSampleNormalizer:
    @staticmethod
    def normalize(x, eps=1e-8):
        # x: [B, T, C, F]
        B, T, C, F = x.shape
        x_flat = x.reshape(B, T * C, F)
        mean = x_flat.mean(dim=1, keepdim=True)
        std = x_flat.std(dim=1, keepdim=True) + eps
        x_norm = (x_flat - mean) / std
        return x_norm.reshape(B, T, C, F), mean, std

    @staticmethod
    def denormalize(x_norm, mean, std):
        return x_norm * std + mean
```

5.2 Data Augmentation

```
class NeuralDataAugmentation:
    def __init__(self, electrode_dropout_prob=0.1, noise_std=0.02):
        self.electrode_dropout_prob = electrode_dropout_prob
        self.noise_std = noise_std

    def electrode_dropout(self, x):
        # Randomly zero out electrodes to simulate missing channels
        mask = torch.rand(x.size(0), 1, x.size(2), 1) > self.electrode_dropout_prob
        return x * mask.float().to(x.device)

    def gaussian_noise(self, x):
        return x + torch.randn_like(x) * self.noise_std

    def __call__(self, x, training=True):
        if not training:
            return x
        if random.random() < 0.5:
            x = self.electrode_dropout(x)
        if random.random() < 0.5:
            x = self.gaussian_noise(x)
        return x
```

6. Training Strategy

6.1 Loss Function

```
class ForecastingLoss(nn.Module):
    def __init__(self, reconstruction_weight=0.3, smoothness_weight=0.05):
        super().__init__()
        self.reconstruction_weight = reconstruction_weight
        self.smoothness_weight = smoothness_weight
        self.mse = nn.MSELoss()

    def forward(self, pred, target, model_outputs=None):
        # Primary loss: prediction MSE
        pred_loss = self.mse(pred, target)
        total_loss = pred_loss

        # Auxiliary: temporal smoothness
        if self.smoothness_weight > 0:
            diff = pred[:, 1:, :] - pred[:, :-1, :]
            smoothness_loss = (diff ** 2).mean()
            total_loss = total_loss + self.smoothness_weight * smoothness_loss

        return total_loss
```

6.2 Optimizer Configuration

Use AdamW with weight decay for regularization, combined with cosine annealing learning rate schedule with 10% warmup.

```
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=1e-3,
    weight_decay=0.01,
    betas=(0.9, 0.98)
)

# Cosine schedule with warmup
total_steps = n_epochs * steps_per_epoch
warmup_steps = int(0.1 * total_steps)

def lr_lambda(step):
    if step < warmup_steps:
        return step / warmup_steps
    progress = (step - warmup_steps) / (total_steps - warmup_steps)
    return 0.5 * (1 + math.cos(math.pi * progress))

scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda)
```

6.3 Early Stopping

Implement early stopping based on validation MSE with patience of 15-20 epochs. Save the best model checkpoint based on validation performance.

7. Implementation Roadmap

7.1 Week 1: Foundation

- **Day 1-2:** Environment setup, download data, run baseline GRU
- **Day 3:** Implement dataset with per-sample normalization
- **Day 4:** Implement FeatureEncoder and positional encoding
- **Day 5:** Implement FactorizedSpatiotemporalBlock
- **Day 6-7:** Assemble basic model, verify training works

7.2 Week 2: Core Features

- **Day 1-2:** Implement iterative refinement
- **Day 3:** Implement multi-component loss
- **Day 4:** Implement data augmentation
- **Day 5:** Training infrastructure (logging, checkpointing)
- **Day 6-7:** Hyperparameter sweep

7.3 Week 3: Refinement

- **Day 1-2:** Analyze predictions, identify failure modes
- **Day 3:** Architecture tweaks based on analysis
- **Day 4:** Train ensemble (3-5 models)
- **Day 5:** Package for challenge submission
- **Day 6-7:** Final tuning and submission

7.4 Priority Matrix

Priority	Component	Impact
P0	Per-sample normalization	High
P0	Factorized attention	High
P1	Iterative refinement	Medium-High
P1	Electrode dropout augmentation	Medium
P2	Multi-component loss	Medium
P3	Ensemble	Medium

8. Hyperparameter Reference

Parameter	Range	Recommended	Notes
d_model	32-128	64	Embedding dimension
n_heads	2-8	4	Must divide d_model
n_layers	2-6	3	Factorized blocks
dropout	0.1-0.3	0.15	Higher for small data
n_refinement	0-3	2	Iterations
batch_size	16-64	32	GPU memory limited
learning_rate	1e-4 to 3e-3	1e-3	With cosine schedule
weight_decay	0.01-0.1	0.01	AdamW

8.1 Search Strategy

- **Stage 1:** Fix d_model=64, n_layers=3. Search lr in [3e-4, 1e-3, 3e-3] and dropout in [0.1, 0.15, 0.2].
- **Stage 2:** With best LR/dropout, try d_model in [32, 64, 128] and n_layers in [2, 3, 4].
- **Stage 3:** Try n_refinement_iters in [0, 1, 2, 3].
- **Stage 4:** Fine-tune around best configuration.

9. Debugging Checklist

9.1 Sanity Checks

- **Overfit one sample:** Can model achieve near-zero loss on a single sample?
- **Gradient norms:** Should be between 1e-5 and 10. Exploding or vanishing = bug.
- **Attention weights:** Temporal attention should show structure, not uniform.
- **Prediction range:** Should match target statistics (mean, std).
- **Ablations:** Removing components should hurt performance.

9.2 Common Failure Modes

Symptom	Likely Cause	Solution
Loss stays high	LR too low/high	Try LR finder
Loss NaN	Numerical instability	Add eps, reduce LR
Train good, val bad	Overfitting	More dropout
Predictions constant	Dead model	Check gradients

9.3 Expected Performance

Model	Relative MSE
Naive (repeat last)	1.0x (baseline)
GRU baseline	0.7-0.8x
This model (basic)	0.5-0.6x
This model (full)	0.4-0.5x
Ensemble	0.35-0.45x

Appendix: Quick Reference Card

Model Architecture
FeatureEncoder(9→64) → SinusoidalPE → [FactorizedBlock×3] → PredHead → Refine×2

Key Hyperparameters
d_model=64, n_heads=4, n_layers=3, dropout=0.15, n_refinement=2, lr=1e-3

Data Flow
Input [B,10,C,9] → Encode [B,10,C,64] → Predict [B,10,C] → Refine → Output

Critical Decisions
- Per-sample normalization (not global)
- No electrode ID embeddings
- Sinusoidal temporal positions
- Factorized attention (not joint)
- Shared feature encoder

Loss
Total = MSE(pred, target) + 0.05×SmoothLoss

Training
AdamW(lr=1e-3, wd=0.01) + Cosine(warmup=10%) + EarlyStopping(patience=15)

Augmentation
ElectrodeDropout(p=0.1) + GaussianNoise(std=0.02)

Notation Reference

Symbol	Meaning	Value
B	Batch size	32
T	Timesteps	10
C	Channels	87/239
F	Features	9
d	Model dim	64
L	Layers	3
K	Refinement iters	2

Remember: Start simple, verify each component, add complexity incrementally. The staged approach ensures you always have a working model to fall back on. Good luck!