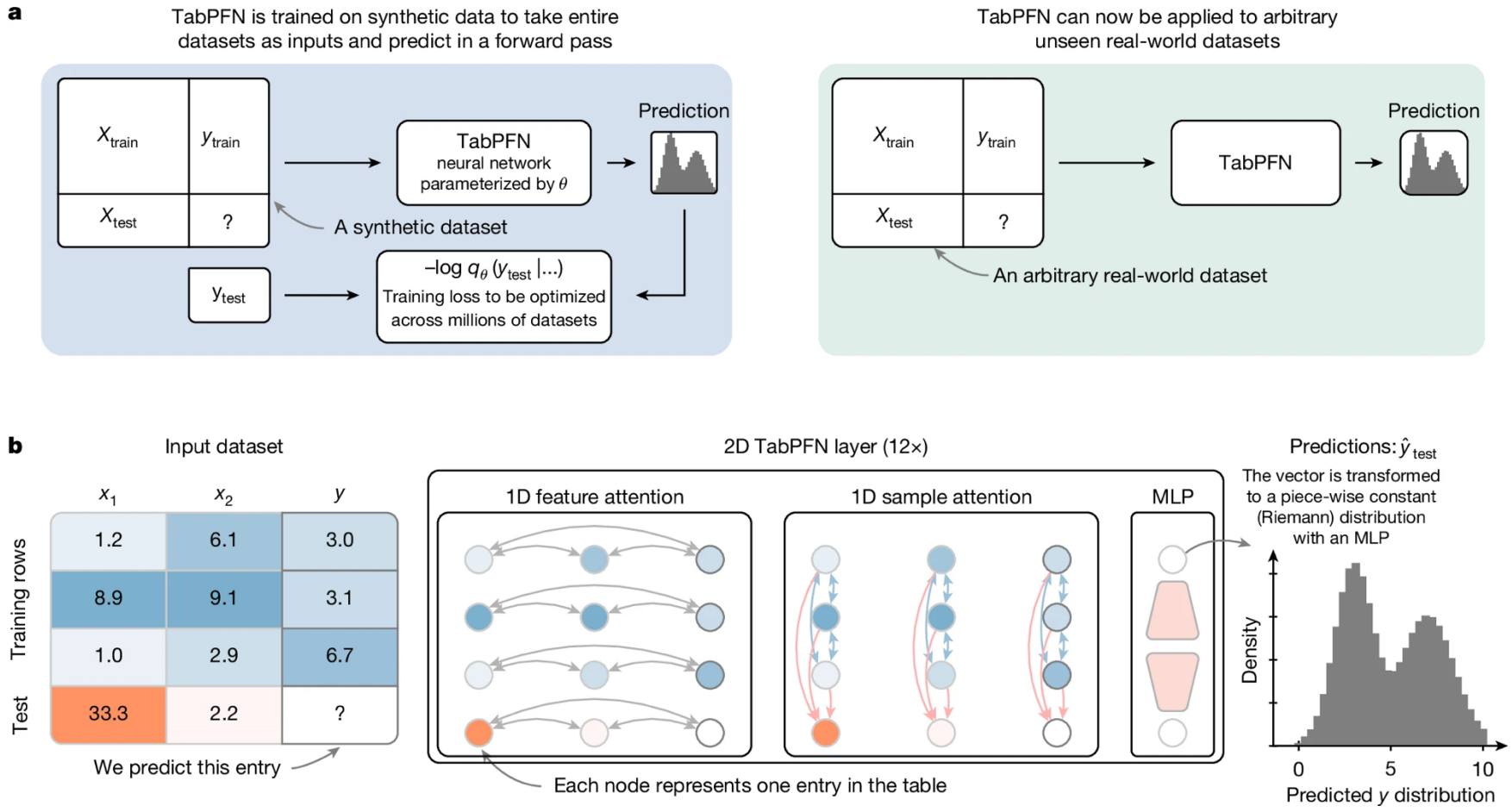
ChunkedTabPFN: Training-free Long Context

Renat Sergazinov
Department of Statistics
Texas A&M University

Shao-An Yin
Department of ECE
University of Minnesota, Twin Cities

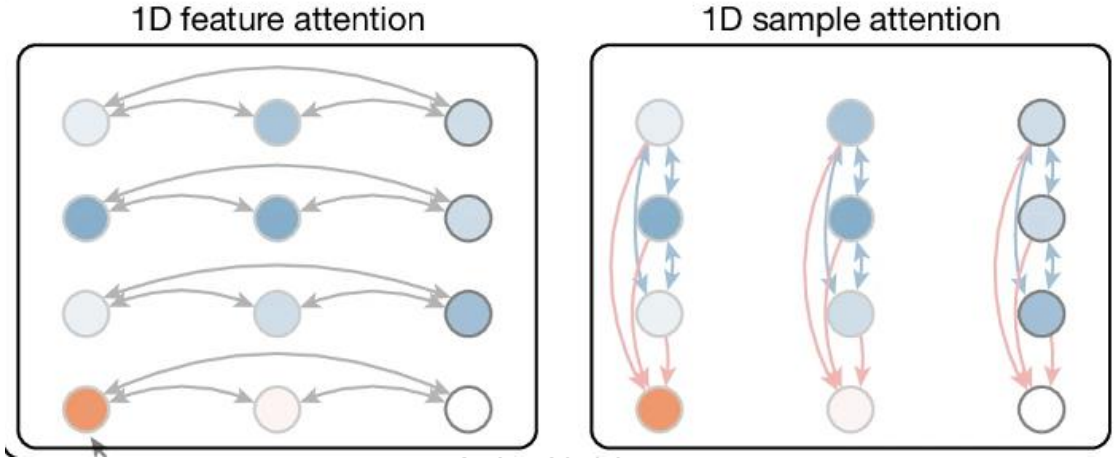
Tabular Foundation Models



* Image courtesy of Hollman et al (2025) “Accurate predictions on small data with a tabular foundation model”, *Nature.*

Issue: Effective Attention Batch Size (B_{eff}) Explodes

- TabPFN employs attention mechanisms both across samples and across features.
  - Feature attention: operates over $(G + 1) \times B$ tokens (across features).
  - Sample attention: operates over $L \times B$ tokens (across samples).
  - Total Cost: $\mathcal{O}\left(\left((G + 1) \times B\right)^2\right) + \mathcal{O}\left((L \times B)^2\right)$.
- Standard LLM attention: no batch expansion; B stays constant.
- With real-world datasets (L > 10K, many features), B_{eff} explodes



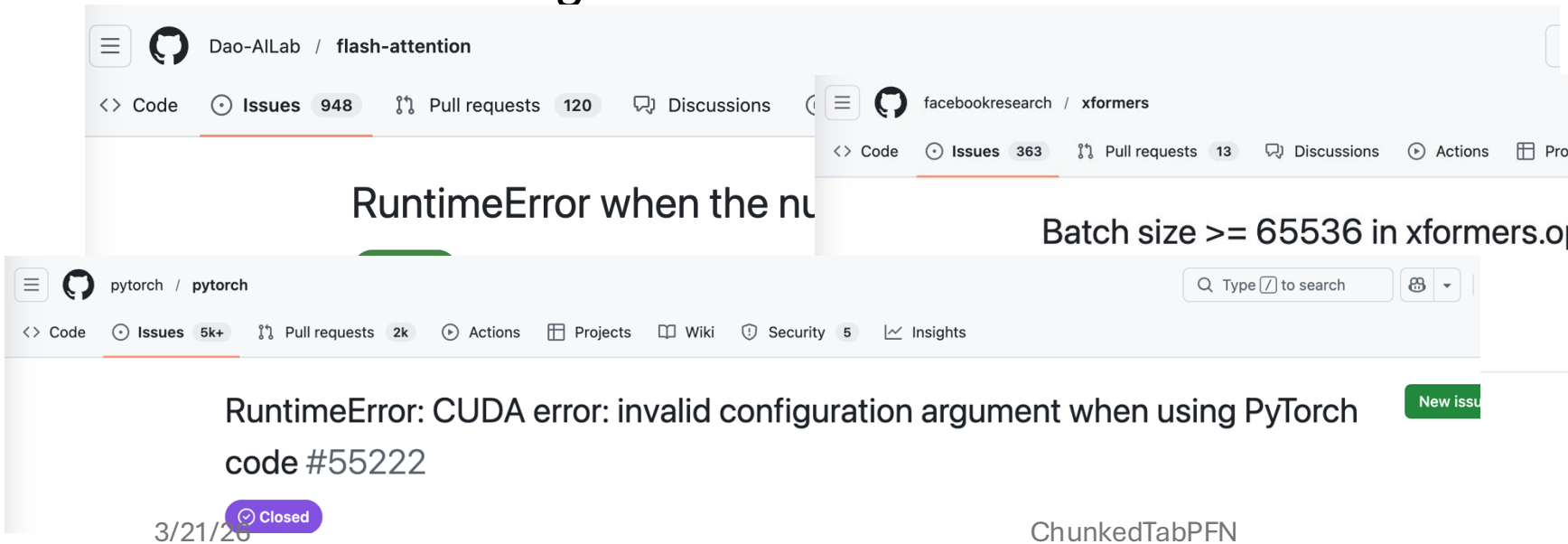
- G: Number of Feature Group.
- L: Number of Samples.
- B: Batch Size.

Two Failure Modes

- **Efficient attention (FlashAttention / xFormers):** CUDA grid limit of 65,535 blocks on Ampere and below

  → invalid configuration argument when $B_{eff} \times H$ exceeds this
- **Math fallback (vanilla PyTorch):** materializes full $L \times L$ attention matrix

  → OOM on large contexts



Our Fix (Part 1): Chunking over B_{eff}

- For GPUs that support efficient attention: chunk the flattened batch dimension so each kernel call stays under the 65,535 limit
- Simple loop over chunks → concatenate outputs

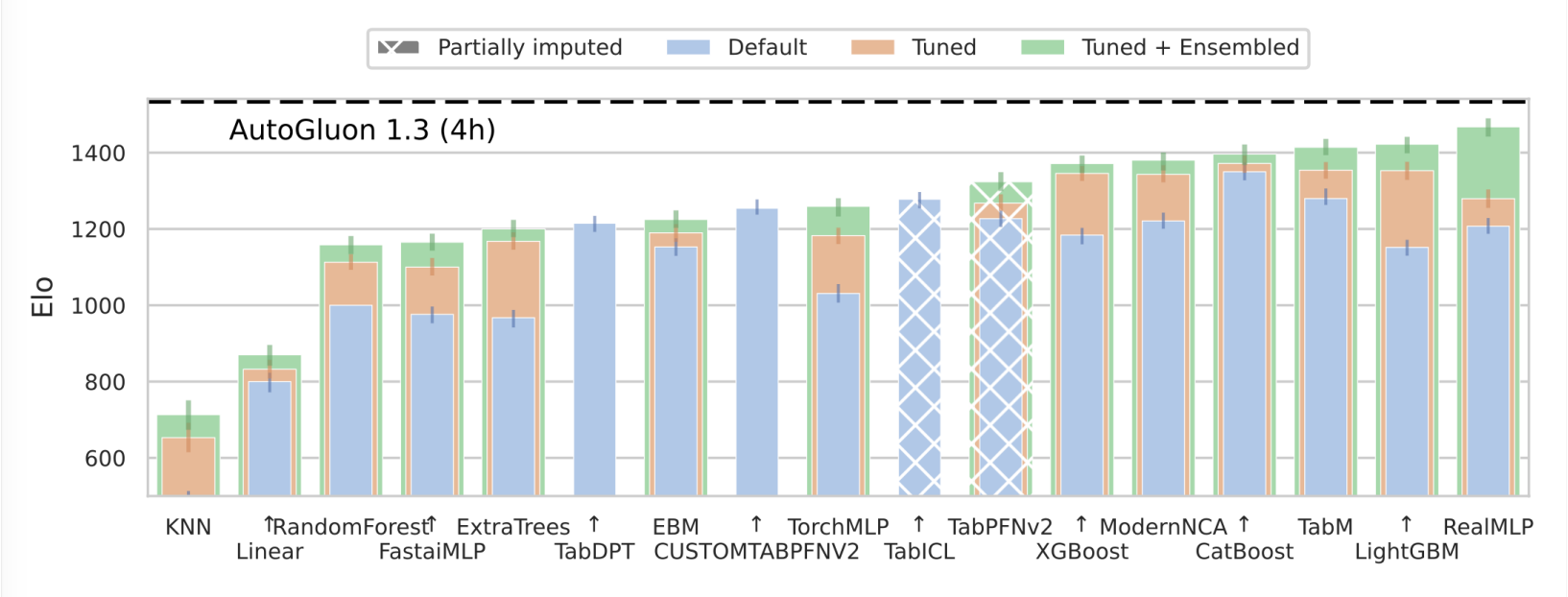
```
for q_chunk, k_chunk, v_chunk in zip(
    torch.split(q, batch_size, dim=0),
    torch.split(k_b, batch_size, dim=0),
    torch.split(v_b, batch_size, dim=0),
):
    # (B_chunk, Lq, H, D) -> (B_chunk, H, Lq, D)
    Q = q_chunk.permute(0, 2, 1, 3).contiguous()
    K = k_chunk.permute(0, 2, 1, 3).contiguous()
    V = v_chunk.permute(0, 2, 1, 3).contiguous()
```

Our Fix (Part 2): Pure-PyTorch Tiled Attention

- For older/consumer GPUs without FlashAttention support
- Tile queries (size l) and keys/values (size r) — only materialize an l × r block at a time
- Maintain running (μ, s, a) statistics via log-sum-exp merge → numerically stable, mathematically exact

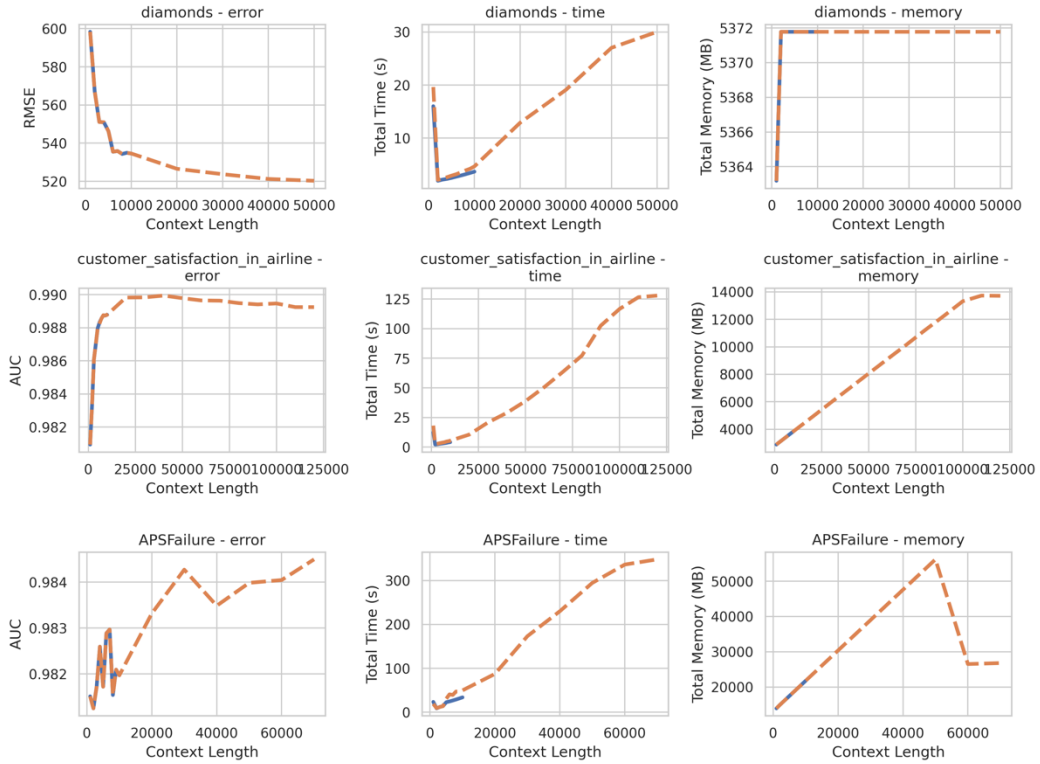
```
for qs in range(0, Lq, q_chunk):
    qe = min(qs + q_chunk, Lq)
    q_tile = q[..., qs:qe, :]                              # (..., l, D)

    # running stats per query row
    mu = q_tile.new_full(q_tile.shape[:-1], -float("inf"))  # (..., l)
    s  = torch.zeros_like(mu)                               # (..., l)
    a  = torch.zeros(*mu.shape, v.size(-1),
                     device=q.device, dtype=q.dtype)        # (..., l, Dv)

    for ks in range(0, Lk, kv_chunk):
```

Results: TabArena Performance



- Figure: Elo / normalized score bar chart (your Figure 2)
- Chunked TabPFN vs. baselines on full TabArena (51 datasets)
- Highlight: prior TabPFN results used Random Forest fallback imputation for OOM datasets — ours runs everything directly

Results: Scaling with Context Length



- Subsampled training sets: 3K → 5K → 10K → 20K → 50K → 100K
- AUC/RMSE continues to improve or plateaus — never degrades

Conclusion

- Chunked TabPFN removes the ~10K sample ceiling → 100K+ on a single 24–32 GB GPU
- Exact computation, no retraining, no hyperparameters, no compression
- Performance keeps improving with more data — the PFN prior is still valuable at large N
- Retains the zero-shot ICL philosophy: just feed in your data

Contact

- If you have any questions, please feel free to contact us.

Renat Sergazinov

Department of Statistics

Texas A&M University

Shao-An Yin

Department of ECE

University of Minnesota, Twin Cities