1. Historical and Conceptual Background

1.1 Why were transformers developed?

Before transformers, sequence modeling was dominated by:

- Recurrent Neural Networks (RNNs),
- Long Short-Term Memory networks (LSTMs),
- Gated Recurrent Units (GRUs).

These models process sequences **step-by-step**, maintaining a hidden state that is updated as new tokens arrive. This has several limitations:

1. **Sequential computation**
   The processing of time index (t) depends on the hidden state from $(t-1)$. You cannot parallelize along the sequence length efficiently.
2. **Long-range dependencies**
   Information about early tokens needs to propagate through many time steps, causing vanishing or exploding gradients. Even with LSTMs/GRUs, learning very long context is hard.
3. **Fixed ordering bias**
   RNNs implicitly encode that the "most recent" tokens are more influential. For some tasks (like bidirectional context in language or global structure in scientific signals), you want a more symmetric view.

In 2017, Vaswani et al. introduced the **Transformer** architecture with the slogan "Attention is all you need". The key ideas:

- Replace recurrence with **self-attention** across all positions.
- Allow **parallel processing** over sequence positions.
- Use **positional encodings** to still encode order.

This led to dramatic improvements in:

- Machine translation,
- Language modeling,
- and eventually, almost every domain where sequences or sets are involved.

1.2 Evolution toward scientific and GW applications

After success in NLP, transformers were adopted in:

- Vision (Vision Transformers),
- Audio and speech,

- Irregular time series,
- Scientific domains (molecular modeling, physics simulations).

For gravitational waves specifically, transformers have been used for:

- Detection (classifying if a GW signal is present),
- Denoising,
- Waveform generation,
- Continuous-wave searches,
- Initial explorations in parameter estimation.

However, **prior GW transformer work** typically did not fully exploit:

- Variable-length input sequences,
- Explicit handling of **missing data** (e.g. missing detectors, frequency gaps),
- Highly structured conditioning in a simulation-based inference (SBI) context.

Dingo-T1's main innovation is to combine **transformers + SBI** in a way that directly addresses:

- Variable detector configurations,
- Variable frequency ranges and cuts,
- Robust posterior estimation across many analysis settings with **one model**.

1a. What is ResNet and Why is it Important?

1a.1 The vanishing gradient problem

Before ResNets, deep neural networks faced a critical problem:

As networks get deeper (more layers), training becomes harder:

- Gradients become very small (vanish) as they backpropagate through many layers.
- Early layers learn very slowly or not at all.
- Counterintuitively, **adding more layers can hurt performance** (even on training data).

This was paradoxical: more capacity should improve expressiveness, but in practice, very deep networks (> 20 layers) performed **worse** than shallow ones.

1a.2 How ResNet solves this: Residual connections

Residual Networks (ResNets), introduced by He et al. (2015), use **skip connections** (also called residual connections or shortcuts):

Instead of learning a direct mapping $\mathcal{F}(x)$, ResNet learns:

$$y = \mathcal{F}(x) + x$$

where: x input to the block.

- $\mathcal{F}(x)$: learned transformation (usually 2-3 conv/FC layers with nonlinearities).
- y: output.

Why this works:

- If the optimal mapping is close to the identity ($y \approx x$), the network only needs to learn $\mathcal{F}(x) \approx 0$ (small corrections).
- Gradients can flow **directly** through the skip connection, bypassing vanishing gradient in $\mathcal{F}$.
- This allows training of networks with **100+ layers** effectively.

1a.3 ResNet internal structure

A typical ResNet block:

1. Input x
2. Apply Conv/FC → BatchNorm → ReLU
3. Apply Conv/FC → BatchNorm
4. Add skip connection:

$$y = \mathcal{F}(x) + x$$

5. Apply ReLU to y

If dimensions don't match (e.g., changing channels), use a **projection shortcut**:

$$y = \mathcal{F}(x) + W_s x$$

where W_s is a learned linear projection.

1a.4 Role of ResNet in Dingo baseline

Dingo baseline (the comparison model) uses a **ResNet-style encoder**:

- Processes multibanded frequency data through residual blocks.
- Compresses

$$d_I(f), S_{n,I}(f)$$

  into a fixed-size feature vector.
- This vector conditions the normalizing flow.

Why ResNet for GW data?

- GW signals span wide frequency ranges → need deep processing.
- Residual connections allow learning hierarchical features (local waveform structure → global chirp pattern).

- Standard choice for fixed-dimensional input before transformers.

Limitation:

- ResNet requires **fixed input size** (specific detectors + frequency range).
- Cannot handle missing detectors or variable frequency ranges without retraining.

1a.5 Why Dingo-T1 moves beyond ResNet

Transformers replace the ResNet encoder in Dingo-T1:

- ResNet: hierarchical, local-to-global, fixed input.
- Transformer: global attention, variable-length, flexible masking.

But ResNets are still powerful for fixed-configuration tasks and serve as a strong baseline.

---

2. High-Level Intuition for Transformers in this Problem

2.1 Why transformers fit GW parameter estimation

In the Dingo-T1 context, the "sequence" is not words in a sentence, but **tokens corresponding to local frequency segments from different detectors**.

Each token encodes:

- A short strain and PSD segment over some frequency interval,
- The frequency bounds of that segment,
- Which detector it came from (H, L, or V).

Transformers are ideal because:

1. **Variable-length input**
   If you remove some segments (e.g. due to detector downtime or frequency cuts), you just have fewer tokens. The transformer does not care about the exact length, only the set/sequence of tokens.
2. **Global context**
   GW waveform parameters correlate information across wide frequency ranges and across detectors:
   - Mass and spin affect the entire chirp.
   - Sky location and distance couple detectors.
     Self-attention lets every token "see" every other token, capturing these global dependencies.
3. **Handling missing data via masking**
   Self-attention already supports masking via an attention mask. By training with masks

that simulate missing detectors and frequency bands, the transformer learns to **marginalize over missing information**.

4. **Set-like behavior**
   The order of tokens (aside from frequency ordering) is not conceptually important. Transformers can behave as powerful set encoders when you include appropriate positional/condition information.

2.2 Data flow through Dingo-T1

At a very high level, the data flow is:

1. **Raw data (**
   ightarrow) frequency domain + PSD
   For each detector, compute $(d_I(f))$ and $(S_{n,I}(f))$.
2. **Multibanding**
   Compress to a non-uniform frequency grid in which the waveform is represented accurately but with fewer bins.
3. **Tokenization**
   Partition the multibanded frequency grid into segments of 16 bins. Each segment + PSD + frequency bounds + detector ID is turned into a token embedding (vector length 1024).
4. **Add summary token**
   Prepend/append a learnable summary token that will aggregate global information through attention.
5. **Apply masks**
   Drop tokens corresponding to:
   - Missing detectors,
   - Unused frequency ranges,
   - Notched bands (calibration issues).
6. **Transformer encoder**
   Pass the sequence of tokens through 8 layers of masked multi-head self-attention + FFN. The summary token collects a global representation.
7. **Projection to context**
   Take the final summary token and linearly project to a 128-dimensional context vector (c)
   .
8. **Conditional normalizing flow**
   Feed (c) into a normalizing flow that models $q(\theta \mid d, S_n)$.
9. **Sampling and importance sampling**
   Draw samples from $q(\theta \mid d, S_n)$, and optionally refine via importance sampling to obtain the final posterior.

2a. Transformer Encoder vs Decoder: Key Distinctions

2a.1 Encoder-only, decoder-only, and encoder-decoder architectures

Transformers come in three main flavors:

1. **Encoder-only** (e.g., BERT, Dingo-T1):
   - Process an input sequence and produce contextualized representations for each token.
   - All tokens attend to all other tokens (bidirectional attention).
   - Used for: classification, feature extraction, embedding tasks.
2. **Decoder-only** (e.g., GPT):
   - Generate sequences autoregressively (one token at a time).
   - Each token only attends to previous tokens (causal masking).
   - Used for: text generation, language modeling.
3. **Encoder-decoder** (e.g., original Transformer for translation):
   - Encoder processes input, decoder generates output.
   - Decoder attends to encoder outputs via **cross-attention**.
   - Used for: sequence-to-sequence tasks (translation, summarization).

2a.2 Why Dingo-T1 uses encoder-only

Dingo-T1 is encoder-only because:

- The task is **regression/density estimation**, not generation.
- We want to extract a **global representation** (context vector) from the entire input.
- No autoregressive structure is needed (we don't generate parameters token-by-token).

The **summary token** acts as the output of the encoder, aggregating information from all input tokens.

2a.3 What about the normalizing flow?

The **normalizing flow** plays a role analogous to a decoder, but:

- It's not a transformer decoder.
- It's a separate probabilistic model that takes the context vector and outputs a distribution over θ.

So the full architecture is:

- **Encoder**: Transformer (tokens → context).
- **"Decoder"**: Normalizing flow (context → posterior distribution).

3. Mathematical Formulation of Attention and Transformers

3.1 Scaled dot-product attention

Given:

- Input tokens $X \in \mathbb{R}^{n \times d_{\text{model}}}$
  (rows are token vectors of dimension (d_{model})),
- Learnable weight matrices W^Q, W^K, $W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$,

we compute:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V.$$

Each row of (Q, K, V) corresponds to a token's query, key, and value.

The **scaled dot-product attention** is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)V$$

Here:

- $M \in \mathbb{R}^{n \times n}$ is the mask matrix:
  - $(M_{ij} = 0)$ if token (j) can be attended to from token (i),
  - $(M_{ij} = -infty)$ if token (j) must be ignored (masked).

The unnormalized attention scores are:

$$s_{ij} = \frac{Q_i \cdot K_j}{\sqrt{d_k}} + M_{ij},$$

and the corresponding attention weights:

$$\alpha_{ij} = \frac{\exp(s_{ij})}{\sum_{j'} \exp(s_{ij'})}.$$

The output for token (i):

$$\text{Attention}(Q, K, V)_i = \sum_j \alpha_{ij} V_j.$$

3.1a Understanding Self-Attention: Intuition and Interpretation

What does "attention" mean intuitively?

Attention is a mechanism for aggregating information based on relevance.

Think of it as a **database query**:

- **Query** (Q): "What am I looking for?"

- **Key** (K): "What does each item represent?"
- **Value** (V): "What information does each item contain?"

For token i:

1. Compute similarity between Q_i (its query) and K_j (keys of all tokens).
2. Convert similarities to weights via softmax (attention distribution).
3. Aggregate values V_j weighted by attention:

$$\sum_j \alpha_{ij} V_j$$

Physical analogy:

- Like a **weighted average** where weights depend on learned relevance.
- Unlike a fixed filter, attention is **data-dependent** and **learned**.

Why "self"-attention?

In **self-attention**:

- Queries, keys, and values all come from the **same input** X.
- Each token attends to **all other tokens** (including itself).

This contrasts with:

- **Cross-attention** (encoder-decoder): queries from decoder, keys/values from encoder.
- **External attention**: queries from input, keys/values from a fixed memory.

For Dingo-T1:

- Tokens are frequency segments and detectors.
- Self-attention allows each segment to "ask": "Which other segments (frequencies, detectors) are most relevant for understanding this signal?"

Example: How attention captures GW correlations

Consider two tokens:

- Token A: Low-frequency segment (20-40 Hz) from Hanford.
- Token B: High-frequency segment (200-300 Hz) from Livingston.

If the true source is a chirping binary:

- Token A's query might "look for" corresponding high-frequency content that matches the chirp evolution.
- The attention score α_{AB} will be **high** if Token B's key indicates it contains the expected merger-frequency content.

- Token A's output aggregates information from Token B (and others) weighted by relevance.

This global aggregation is why transformers excel at capturing long-range correlations.

3.2 Multi-head self-attention

Instead of one attention head, transformers use (h) heads:

For head (ell):

$$Q^{(\ell)} = XW^{Q,(\ell)}, \quad K^{(\ell)} = XW^{K,(\ell)}, \quad V^{(\ell)} = XW^{V,(\ell)}$$

Compute attention per head:

$$\mathrm{head}^{(\ell)} = \mathrm{Attention}\Big(Q^{(\ell)}, K^{(\ell)}, V^{(\ell)}\Big).$$

Concatenate heads along the feature dimension and project:

$$\mathrm{MultiHead}(X) = \mathrm{Concat}\big(\mathrm{head}^{(1)}, \ldots, \mathrm{head}^{(h)}\big)W^O$$

Intuitively:

- Each head can specialize in different frequency ranges, detectors, or aspects of the signal/noise structure.
- Multi-head attention allows richer representations.

3.2a Multi-Head Attention: NOT Multiple Detectors!

Common misconception:
"Does each head correspond to one detector (H, L, V)?"

Answer: NO.

What multi-head actually means

Number of heads ($h = 16$ in Dingo-T1) refers to:

- **16 parallel attention mechanisms** within each transformer layer.
- Each head has its own

  $$W^Q, W^K, W^V$$

  matrices.
- Each head learns to attend to **different aspects** of the data.

Not detector-specific:

- Heads are **not** assigned to detectors.
- All tokens (from all detectors) are processed by **all heads**.

Why multiple heads?

Multi-head attention allows the model to:

- Attend to different **frequency correlations** simultaneously.
- Capture both **local** (within-detector) and **global** (cross-detector) patterns.
- Learn multiple **representational subspaces** (e.g., one head for amplitude, another for phase).

Analogy:

- Like having **multiple experts** looking at the same data from different perspectives.
- Each expert (head) produces an opinion (attention-weighted output).
- Outputs are combined (concatenated and projected) to form a comprehensive representation.

Example: What might different heads learn?

In Dingo-T1, heads might specialize:

- **Head 1**: Correlations between low and high frequencies (chirp pattern).
- **Head 2**: Cross-detector timing information (time delays encode sky position).
- **Head 3**: PSD-dependent weighting (which frequencies are noisy vs clean).
- **Head 4**: Precession signatures (spin-induced modulation).

These specializations **emerge during training** and are not pre-assigned.

3.3 Transformer encoder layer (pre-LN)

Each Dingo-T1 encoder layer uses **pre-layer normalization**:

Let $X^{(l)}$ be the input to layer l:

1. Self-attention sub-layer:

$$Y^{(l)} = X^{(l)} + \mathrm{MultiHead}\big(\mathrm{LN}(X^{(l)})\big).$$

2. Feed-forward network (FFN) sub-layer:

$$Z^{(l)} = Y^{(l)} + \mathrm{FFN}\big(\mathrm{LN}(Y^{(l)})\big).$$

Where LN is LayerNorm over the feature dimension, and FFN is:

$$\mathrm{FFN}(x) = W_2\,\sigma(W_1 x + b_1) + b_2,$$

with hidden size 2048 and nonlinearity σ (e.g. GeLU/ReLU).

There are $N = 8$ such layers in Dingo-T1.

3.3a Feed-Forward Networks (FFN): Structure and Role

What is the FFN block?

After self-attention, each token passes through a **position-wise feed-forward network**:

$$\mathrm{FFN}(x) = W_2\,\sigma(W_1 x + b_1) + b_2$$

where:

- $W_1 \in \mathbb{R}^{d_{\mathrm{model}} \times d_{\mathrm{ff}}}$: expands dimension (1024 → 2048).
- σ: nonlinearity (ReLU, GeLU).
- $W_2 \in \mathbb{R}^{d_{\mathrm{ff}} \times d_{\mathrm{model}}}$: projects back (2048 → 1024).

Key point: FFN is applied **independently** to each token (no mixing across tokens).

Why follow attention with FFN?

Self-attention does:

- **Linear** mixing of value vectors V_j (weighted average).
- No nonlinear transformation of individual token features.

FFN provides:

- **Nonlinear processing** of each token's representation.
- Capacity to transform features in a token-specific way.

Together:

- **Attention**: aggregates information **across tokens**.
- **FFN**: processes information **within each token**.

This alternation is crucial for learning complex functions.

Mathematical structure

The FFN is a **two-layer MLP with one hidden layer**:

1. Expand:

   $$h = \sigma(W_1 x + b_1)$$

   (dimension increases).
2. Compress:

   $$y = W_2 h + b_2$$

   (dimension returns to d_{model}).

The **expansion factor** ($d_{\mathrm{ff}}/d_{\mathrm{model}} = 2048/1024 = 2$) is standard in transformers.

Role in Dingo-T1

For GW data:

- After attention aggregates frequency and detector information, FFN:
  - **Refines** each token's representation.
  - **Encodes** nonlinear relationships (e.g., amplitude vs frequency, PSD-dependent weighting).
  - **Prepares** features for the next layer's attention.

Without FFN, the model would be a stack of linear operations (since attention without nonlinearity is linear), severely limiting expressiveness.

3.4 Masking through M

Dingo-T1 exploits the mask M to represent missing tokens:

- When a token (k, I) (segment k, detector I) is masked, it is simply **removed from the input sequence**; effectively, the sequence is shorter.
- For tokens that remain, $M_{ij} = 0$ for all pairs (i, j). There is no causal masking since this is an encoder-only architecture (no autoregressive constraint).

Alternatively, you can think of masked tokens as present in the sequence but with:

- Their rows zeroed,
- Or their positions set to be un-attendable via $M_{ij} = -\infty$.

The main effect: **only unmasked segments contribute** to the context vector that conditions the flow.

---

4. Architecture Details in Dingo-T1

4.1 Tokenizer: from segments to embeddings

Each segment of 16 multibanded frequency bins is mapped via:

- A fully-connected layer,
- A 512-dimensional residual block,
- Conditional injection of frequency bounds and detector ID via a gated linear unit (GLU).

Let the raw features for token (j) be:

- Strain $(d^{(j)} \in \mathbb{R}^{32})$ (16 complex bins as real+imag),
- PSD $(S_n^{(j)} \in \mathbb{R}^{16})$,
- Frequency bounds $((f_{min}^{(j)}, f_{max}^{(j)}) \in \mathbb{R}^2)$,

- Detector one-hot ID $(e_I^{(j)} \in \mathbb{R}^3)$???.

We can denote the tokenizer as:

$$t_j = \text{Tokenizer}\big(d^{(j)}, S_n^{(j)}, f_{min}^{(j)}, f_{max}^{(j)}, e_I^{(j)}\big) \in \mathbb{R}^{1024}.$$

The tokenizer is **shared** across all detectors; the detector identity is part of the conditional input.

4.2 Summary token

We define a learnable vector:

$$s_0 \in \mathbb{R}^{1024},$$

initialized randomly and updated during training. This is concatenated with the $(3K)$ data tokens:

$$X_0 = \begin{bmatrix} s_0 \\ t_1 \\ \vdots \\ t_{3K} \end{bmatrix} \in \mathbb{R}^{(3K+1)\times 1024}.$$

After passing through the transformer encoder, we obtain:

$$X_N = \text{TransformerEnc}(X_0),$$

and we extract the final summary token:

$$s_N = X_N[0,:] \in \mathbb{R}^{1024}.$$

4.3 Projection to context

We compute a 128-dimensional context vector:

$$c = W_c s_N + b_c, \quad c \in \mathbb{R}^{128}.$$

This is the **only** input from the transformer into the normalizing flow; token-level details are summarized here.

4.4 Conditional normalizing flow

The flow models the conditional posterior $(q(\theta \mid c))$:

- Base latent $(z \sim \mathcal{N}(0, I))$.
- Apply a sequence of conditional coupling layers with rational-quadratic splines, whose parameters depend on both:
  - Parts of (θ),
  - The context vector(c).

Each coupling layer is invertible and has a tractable Jacobian determinant, allowing exact densities and sampling.

4a. How Transformers Produce Posterior Distributions

4a.1 The full pipeline: tokens → posterior

Step-by-step process:

1. **Input**: GW strain data d, PSD S_n → tokenized to $X_0 \in \mathbb{R}^{n\times 1024}$.
2. **Transformer encoder**: Processes X_0 through 8 layers to produce X_N.
3. **Extract summary**: $s_N = X_N[0,:]$ (the summary token after final layer).
4. **Project to context**: $c = W_c s_N + b_c \in \mathbb{R}^{128}$.
5. **Normalizing flow**: Models $q(\theta \mid c)$, a probability density over parameters θ.

Key insight:

- The transformer does **not** directly output posterior samples.
- It outputs a **context vector** c that **summarizes the data**.
- The **normalizing flow** uses c to define a flexible distribution over θ.

4a.2 Physical intuition: compression and conditioning

**What does the context vector c represent?

c is a **low-dimensional summary** of the high-dimensional data d, S_n:

- Contains information about:
  - Mass range (from chirp frequency evolution).
  - Distance (from amplitude).
  - Spins (from precession, merger details).
  - Sky location (from detector time delays, amplitude ratios).
- Discards noise and irrelevant details.

Analogy:

- The transformer is like an **expert analyzer** who reads all the data and writes a brief report (c).
- The normalizing flow is like a **probabilistic model** that, given the report, outputs a distribution of plausible parameter values.

4a.3 Mathematical derivation: from c to $p(\theta \mid d)$

Training objective:

The model is trained to minimize:

$$\mathcal{L} = \mathbb{E}_{p(\theta,d,S_n)}\left[-\log q(\theta \mid c(d, S_n))\right]$$

where $c(d, S_n)$ is the transformer's output context.

This is equivalent to **maximizing the likelihood** of the true parameters under the neural model.

At inference:

Given observed data $d_{\text{obs}}, S_{n,\text{obs}}$:

1. Compute

$$c_{\text{obs}} = \text{Transformer}(d_{\text{obs}}, S_{n,\text{obs}})$$

2. Sample from $q(\theta \mid c_{\text{obs}})$ using the normalizing flow.

Why this works:

By training on millions of (θ, d, S_n) samples:

- The transformer learns to extract **sufficient statistics** for θ from d, S_n.
- The flow learns to map those statistics to the correct posterior shape.

4a.4 Practical implementation logic (pseudo-code)

```
# Training
for epoch in range(num_epochs):
    for batch in data_loader:
        theta, d, Sn = sample_simulation_data(batch_size)

        # Tokenize and mask
        tokens = tokenizer(d, Sn)
        masks = sample_masks()
        tokens_masked = apply_masks(tokens, masks)

        # Transformer encodes
        context = transformer_encoder(tokens_masked)  # shape: [B, 128]

        # Flow evaluates log probability
        log_q = flow.log_prob(theta, context)  # shape: [B]

        # Loss: negative log-likelihood
        loss = -log_q.mean()

        # Optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```python
# Inference
def infer(d_obs, Sn_obs):
    tokens = tokenizer(d_obs, Sn_obs)
    context = transformer_encoder(tokens)
    samples = flow.sample(context, num_samples=100000)
    return samples
```

4a.5 Why this is better than traditional methods

Traditional samplers (MCMC, nested sampling):

- Evaluate likelihood

$$p(d \mid \theta)$$

  millions of times.
- Each evaluation requires waveform generation (expensive).
- Takes **hours to days** per event.

Dingo-T1:

- Waveform generation happens once during **training** (millions of simulations).
- At inference: **single forward pass** through transformer + flow.
- Takes **seconds** per event.

The transformer acts as a **learned compressor** that bypasses expensive likelihood evaluations.

---

5. Training Dynamics and Optimization (Transformers + Flow)

5.1 Joint training

The tokenizer, transformer encoder, and normalizing flow are trained **jointly** end-to-end, with loss:

$$\mathcal{L} = \mathbb{E}_{p(\theta)p(S_n)p(d|\theta,S_n)p(m)}\big[-logq(\theta \mid m(d), m(S_n))\big].$$

This is a standard negative log-likelihood (NLL) objective for conditional density estimation.

- Gradients propagate:
  - From flow parameters (phi) into context (c),
  - Through transformer weights,
  - Through the tokenizer.

The optimization learns:

- How to encode GW data into a context vector that is maximally informative about (θ) under masking.
- How to parameterize the conditional posterior $(q(\theta \mid c))$ flexibly and accurately.

5.2 Effect of masking during training

Sampling mask $(m \sim p(m))$ changes the input to:

- $(m(d))$: some segments removed (detectors/frequencies).
- $(m(S_n))$: corresponding PSD segments removed.

The network thus sees a distribution of **incomplete data** and learns to optimize NLL under that distribution. Intuitively:

- For configurations with missing detectors, the model must rely more heavily on remaining detectors and adjust uncertainty.
- For configurations with narrower frequency ranges, the model learns to use only partially observed waveform information.

Because the flow sees many masked configurations for the same underlying (θ) and (S_n), it essentially learns to approximate:

$$q(\theta \mid m(d), m(S_n)) \approx p(\theta \mid m(d), m(S_n)).$$

Over training, the transformer becomes a **universal encoder** over the space of masking patterns defined by $(p(m))$.

5.3 Optimization details

Key choices that impact training dynamics:

- **AdamW** with $(\beta_1 = 0.8, \beta_2 = 0.99)$ and weight decay (0.005).
- Large batch size (16,384) to stabilize gradient estimates for flows and transformers.
- Pre-LN transformer to avoid unstable gradients.
- Learning rate schedule that halves LR on plateaued validation loss (adaptively reduces step size as training converges).

Training times (9–12 days) reflect:

- The heavy cost of simulating waveforms and flows.
- The need to cover a wide range of (θ), (S_n), and mask patterns for amortization.

5a. Transformer Philosophy: Differences from CNNs and Traditional NNs

5a.1 Three paradigms for neural architectures

Aspect	Traditional NN (MLP)	CNN (ResNet)	Transformer (Dingo-T1)
Input structure	Fixed-size vector	Grid (image, fixed freq. grid)	Sequence/set of tokens
Processing	Fully connected layers	Local convolutions + pooling	Global self-attention
Receptive field	Immediate (all-to-all)	Local, grows with depth	Global from layer 1
Inductive bias	None (very general)	Locality, translation equivariance	Permutation equivariance (with positional info)
Flexibility	Fixed input size	Fixed input size	Variable sequence length
Parallelization	Across samples only	Across spatial positions	Across tokens (sequence length)

5a.2 Strengths of each approach

Traditional NNs (MLPs):

- **Strengths**: Universal approximators, simple to implement.
- **Weaknesses**: No structural assumptions → require huge amounts of data; scale poorly to high dimensions.

CNNs (ResNets):

- **Strengths**: Excellent for **grid-structured data** (images, fixed spectrograms). Exploit locality and translation symmetry.
- **Weaknesses**: Fixed input size; poor at long-range dependencies (need many layers); cannot handle variable-length or missing data easily.

Transformers:

- **Strengths**: Handle **variable-length sequences**; capture **global dependencies** immediately; flexible to missing data (masking).
- **Weaknesses**: Require **more data and compute** to train (quadratic complexity in sequence length); less built-in inductive bias (more general but need more data to learn structure).

5a.3 Why transformers for gravitational-wave inference?

GW data characteristics:

- **Variable configurations**: detectors go offline, frequency ranges change.
- **Long-range correlations**: chirp spans entire frequency range; detectors are coupled by sky position.

- **Not naturally grid-like**: multibanded frequency is non-uniform; detectors are discrete, not spatially ordered.

Transformers match these needs:

- **Tokenization** converts heterogeneous data (different detectors, non-uniform frequencies) into a unified sequence representation.
- **Self-attention** captures correlations across all frequencies and detectors without needing deep hierarchies.
- **Masking** naturally handles missing detectors and frequency cuts.

CNNs would struggle because:

- Fixed-size input → separate model for each detector combination.
- Locality bias → need many layers to correlate low and high frequencies.
- No native support for variable-length or missing segments.

5a.4 Fundamental differences in how they learn

CNNs learn:

- **Hierarchical features**: edges → textures → objects.
- **Translation-invariant patterns**: same filter applied everywhere.

Transformers learn:

- **Context-dependent relationships**: what matters depends on the full input (e.g., this frequency matters *because* that detector shows a specific pattern).
- **Adaptive aggregation**: attention weights are data-dependent, not fixed filters.

For GW parameter estimation:

- The **optimal way to combine information** depends on which detectors are present, their noise levels, and the signal's frequency content.
- Transformers can **learn these dependencies** from data, while CNNs would need hand-engineered preprocessing or architectural tricks.

---

6. Architecture Summary (Tabular)

A compact summary of the Dingo-T1 ML pipeline:

Stage	Input	Output	Model type
Data prep	Raw strain $(d_I(t))$, PSD	$(d_I(f), S_{n,I}(f)$)	FFT + Welch PSD

Stage	Input	Output	Model type
Multibanding	$(d_I(f), S_{n,I}(f))$	Multibanded $(d_I, S_{n,I})$	Deterministic compression
Tokenization	Local segments (16 bins)	Tokens $(t_j \in \mathbb{R}^{1024})$	Shared MLP + residual + GLU
Sequence build	Tokens (H, L, V) + summary	$(X_0 \in \mathbb{R}^{208\times1024})$	Concatenation
Masking	(X_0), PSD tokens	Subsequence + mask (M)	Stochastic token dropping
Transformer enc.	Tokens + mask	Summary token (s_N)	8-layer encoder, 16 heads (pre-LN)
Projection	Summary token	Context $(c \in \mathbb{R}^{128})$	Linear layer
Flow	Context, latent (z)	Posterior $(q(\theta \mid c))$	Conditional rational-quadratic spline flow
IS (optional)	Samples from (q)	Weighted samples $(simp(\theta \mid d))$	Importance sampling with true likelihood

---

7. Pseudocode View: Practical Implementation

Below is conceptual pseudocode (Python-like) illustrating the Dingo-T1 pipeline for **training** and **inference**.

7.1 Training pseudocode

```
# Pseudocode — high-level, not runnable as-is

for epoch in range(num_epochs):
    for batch in data_loader:  # batch size B
        # 1. Sample parameters and generate data
        theta_batch = sample_prior(B)                           # θ ~ p(θ)
        Sn_batch = sample_PSDs(B)                               # PSD ~
p(Sn)
        d_batch    = simulate_waveforms(theta_batch, Sn_batch) # d ~ p(d|
θ,Sn)

        # 2. Multibanding
        d_mb, Sn_mb = multiband(d_batch, Sn_batch)

        # 3. Tokenization
        tokens = tokenizer(d_mb, Sn_mb)  # shape: [B, num_tokens, d_model]

        # 4. Sample mask and apply (data-based masking)
```

~~~
        masks = sample_masks(B, tokens)  # struct specifying which tokens
drop
        tokens_masked, Sn_masked = apply_masks(tokens, Sn_mb, masks)

        # 5. Transformer encoder
        summary = transformer_encoder(tokens_masked)  # [B, d_model]
        context = context_linear(summary)             # [B, 128]

        # 6. Conditional normalizing flow
        # Flow provides log q(θ | context) for each θ in batch
        log_q = flow.log_prob(theta_batch, context)   # [B]

        loss = -log_q.mean()

        # 7. Optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step_if_needed(loss)
7.2 Inference pseudocodedef infer_posterior(d_obs, Sn_obs,
detector_config, freq_config):
    # d_obs, Sn_obs: observed data/PSD for one event
    # 1. Multibanding
    d_mb, Sn_mb = multiband(d_obs, Sn_obs)

    # 2. Tokenization
    tokens = tokenizer(d_mb, Sn_mb)

    # 3. Build mask from desired config (detectors + freq cuts)
    mask = build_inference_mask(tokens, detector_config, freq_config)
    tokens_masked, Sn_masked = apply_masks(tokens, Sn_mb, mask)

    # 4. Transformer encoder -> context
    summary = transformer_encoder(tokens_masked)
    context = context_linear(summary)

    # 5. Flow samples
    theta_samples, log_q = flow.sample_and_log_prob(context,
num_samples=1e5)

    # 6. Importance sampling in uniform frequency domain
    log_p = true_log_likelihood_and_prior(theta_samples, d_obs, Sn_obs)
    log_w = log_p - log_q  # up to constant
    w = normalize_weights(log_w)

    return theta_samples, w
    ```

This illustrates how a single trained model can be reused across events
with varying configurations simply by changing the inference mask.
~~~

```markdown
---

## 8. How and Why Transformers Are Relevant Specifically Here

To connect back to the original **Phase-2 goals**:

### Historical context
Transformers were introduced to overcome RNN limitations and enable fully
parallel sequence processing using attention.
Dingo-T1 applies these advantages to GW signals represented as sequences
of frequency-domain segments.

---

### Conceptual understanding
- Tokens represent local “views” of the signal (frequency segments from
each detector plus PSD).
- Self-attention learns which segments matter most for a given detector
combination and frequency configuration.
- The summary token functions as a learnable global statistic of the
entire multi-detector, multi-frequency input.

---

### Mathematical formulation
We described attention as:
$$
\text{Attention}(Q, K, V)
= \text{softmax}\!\left( \frac{Q K^\top}{\sqrt{d_k}} + M \right) V
$$
including the masking term \(M\) that enforces missing data.

- The transformer encoder layers are **pre-LN** stacks of multi-head
attention + FFN, enabling stable training.
- The conditional normalizing flow uses **rational-quadratic splines** to
provide an expressive posterior model with tractable densities.

---

### Architecture details
- Embedding dimension: $(d_{\text{model}} = 1024)$
- Encoder depth: 8 layers
- Attention heads: 16
- FFN width: 2048
- Tokens are of fixed size (16 frequency bins), but the number of tokens
varies with:
  - included detectors
  - selected frequency ranges
- Data-based masking ensures that during training, the transformer sees
```

```markdown
the same distribution of configuration variations as encountered during
inference.

---

### Coding perspective
The implementation closely follows standard transformer encoder
architectures (e.g. `torch.nn.TransformerEncoder`), with:
- custom tokenization,
- custom masking logic,
- a conditional normalizing-flow head.

The main GW-specific components are:
- Multibanding
- Tokenization with PSD and detector identity
- Data-based masking strategies
## 9. Summary and Bridge to Phase 3

This Phase 2 report has covered:
- **ResNet**: What it is, how residual connections work, role in baseline
Dingo.
- **Transformers**: Historical context, encoder vs decoder, mathematical
foundations of attention.
- **Self-attention**: Intuition, math, multi-head (NOT detectors!).
- **FFN**: Structure, role in processing token representations.
- **Transformer philosophy**: Fundamental differences from CNNs and
traditional NNs.
- **Posterior production**: How transformers + flows yield distributions.
- **Implementation**: Pseudocode for training and inference.

**Phase 3** will focus on:
- Data generation, cleaning, and preprocessing.
- Training datasets and evaluation metrics.
- Interpretation of results (violin plots, corner plots, sample
efficiency).
- Code structure and reproducibility.

All architectural concepts established here will be applied to understand
how the model performs on real and simulated data.
```