05 Attention and Transformers

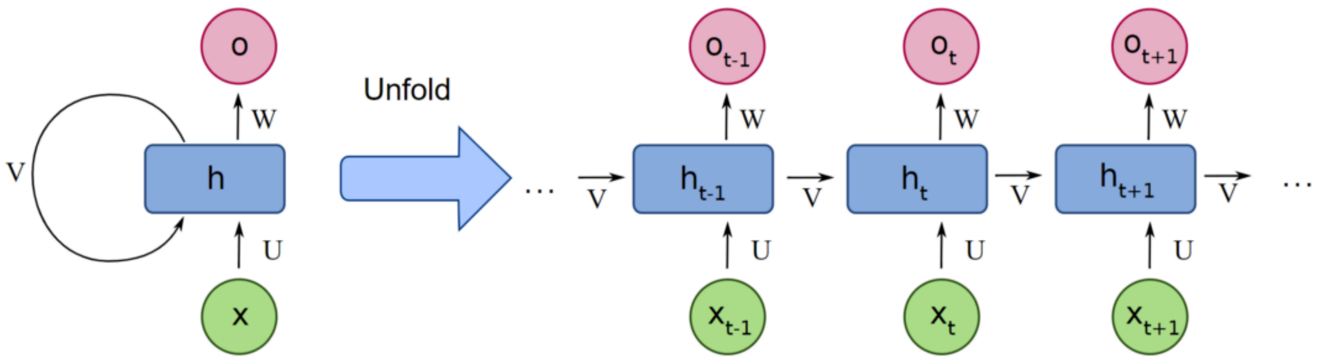
Today's Roadmap

1. **From RNNs to Attention**: The motivation
2. **The Attention Mechanism**: Core concepts and mathematics
3. **Transformer Architecture**: Building blocks
4. **Why Transformers Work**: Theoretical insights
5. **Sequential Data Representation**: Beyond NLP
6. **Architecture Deep Dive**: Implementation details
7. **Computational Considerations**: Efficiency and scaling

---

1. The Problem with Recurrent Architectures

The Sequential Bottleneck




```
RNN/LSTM Processing:
 h₁ → h₂ → h₃ → h₄ → ... → hₙ
 ↑    ↑    ↑    ↑          ↑
 x₁   x₂   x₃   x₄         xₙ

Problems:
- Sequential computation (can't parallelize)
- Long-range dependencies vanish
- Information bottleneck through hidden state
```

Training RNN

1. training translates to *expansion* of the recursive structure to a multilayer perceptron
2. now we can use gradient descent algorithm
3. $$L(x_1,\dots,x_T,y_1,\dots,y_T,w_h,w_o)=\frac{1}{T}\sum_{t=1}^{T}l_t(y_t,o_t)$$
$$\frac{\partial L}{\partial w_h}=\frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t,o_t)}{\partial w_h}=\frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t)}{\partial o_t}\frac{\partial g(h_t,w_o)}{\partial h_t}\frac{\partial h_t}{\partial w_h}$$
$$\frac{\partial h_t}{\partial w_h}=\frac{\partial f(x_t,h_{t-1},w_h)}{\partial w_h}+\frac{\partial f(x_t,h_{t-1},w_h)}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial w_h}$$
4. last $\frac{\partial h_t}{\partial w_h}$ (last part) on the weights from the last state h_{t-1} and, at the same time w_h (first part after equation)
5. it is possible to make some approximations, but it is still not a stable computation
6. it needs to be computed only on some last T states only

The Vanishing Gradient Problem Revisited

Even with LSTM/GRU:

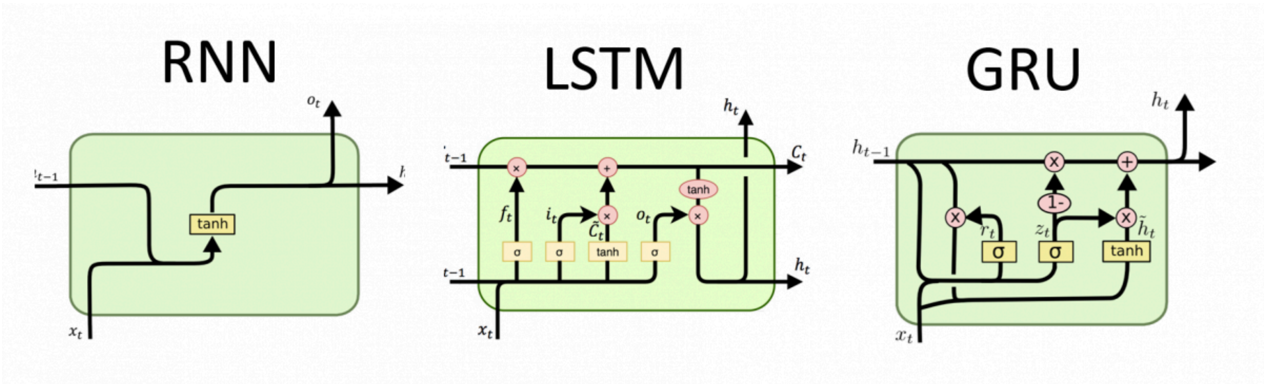
- Path length between distant tokens grows linearly: $O(n)$
- Gradient flow diminishes over long sequences
- Earlier tokens have exponentially smaller influence

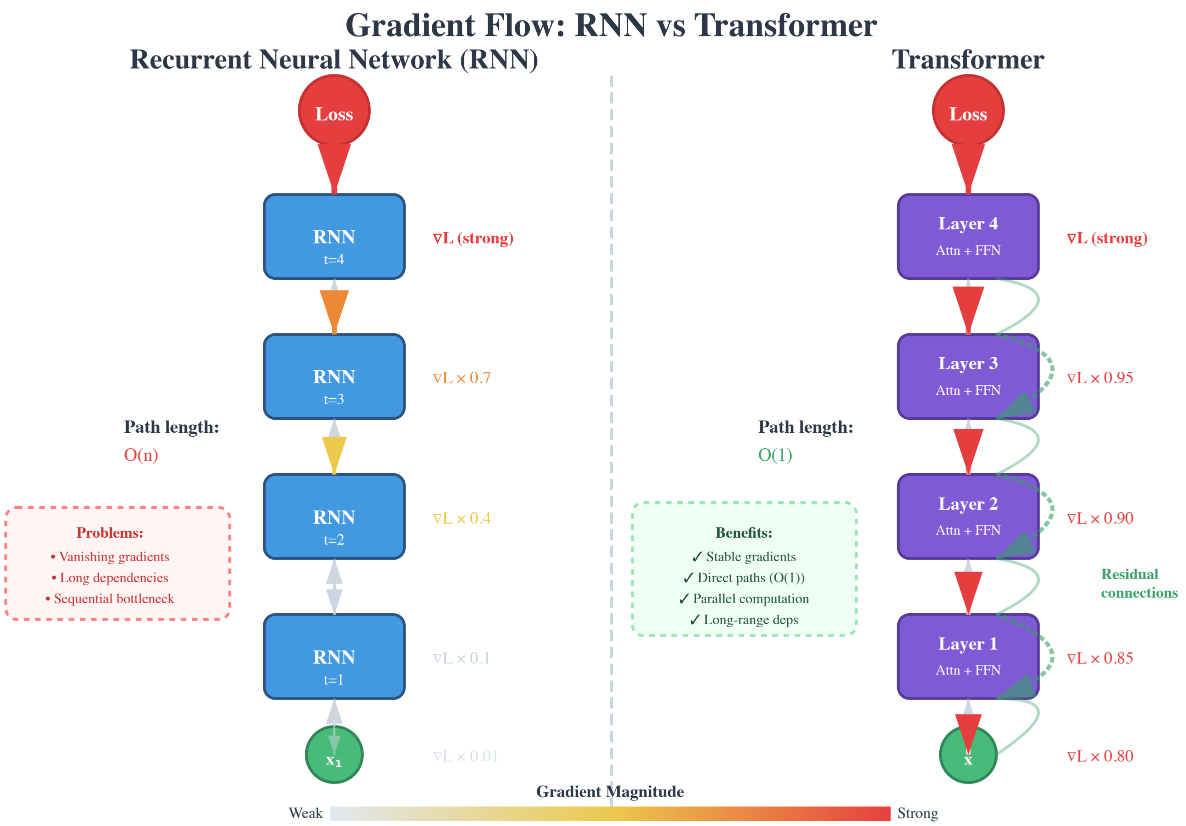
Mathematical View:

$$\frac{\partial h_t}{\partial h_0} = \prod_{i=1}^{t} \frac{\partial h_i}{\partial h_{i-1}}$$

If $\left|\frac{\partial h_i}{\partial h_{i-1}}\right| < 1$, gradients vanish as $t \to \infty$

- partially solved in recurrent models like **GRU** and **LSTM**
  - introduction of ReLU activation
  - gating





- RNN
  - sequential flow
  - diminishing gradient
  - long path with no shortcuts
- Transformer
  - flow by layers with residual connections

- stable gradients with slight degradation

The Fixed Context Problem

Encoder-Decoder RNN:

```
# Entire input compressed into single vector c
encoder_output = encode(x₁, x₂, ..., xₙ)  # → c (fixed size!)
decoder_output = decode(c, y₁, y₂, ..., yₘ)

# Problem: c must capture EVERYTHING about the input
# This is an information bottleneck!
```

“rnn-encoder-decoder.svg” could not be found.

Example: Translation

```
English: "The agreement on the European Economic Area was signed in August 1992"
         [14 words → compressed to fixed vector c → decode]

For each output word, decoder sees:
- Same context vector c
- No direct access to specific input words
- Can't "look back" at relevant parts
```

---

2. Attention

> "Instead of encoding the entire input into a fixed context vector, let the decoder **attend** to different parts of the input at each decoding step."

The Problem: Fixed Context Bottleneck

- before attention
  - encoder-decoder compressed the **entire input sequence into a** single fixed-size context vector**
  - e.g. *"The agreement on the European Economic Area was signed in August 1992"*
  - all 14 words had to be squeezed into one vector
  - severe information bottleneck, especially for long sequences
  - loosing important relations between words

The Bahdanau Innovation

[Bahdanau et al. (2014)](#) (over 40 thousand citations) first attention mechanism for neural machine translation.

- instead of using a fixed context vector,
- decoder **dynamically attends to different parts of the encoder output** at each decoding step:

1. **Encoder produces a sequence of hidden states**: $h1, h2, \ldots, h_n$ (one per input word)
2. **At each decoder step t**, compute **alignment scores** between decoder state s_{t-1} and each encoder state

$$h_j : e_{tj} = \text{score}(s_{t-1}, h_j) = v^T \tanh(W_1 s_t + W_2 h_j)$$

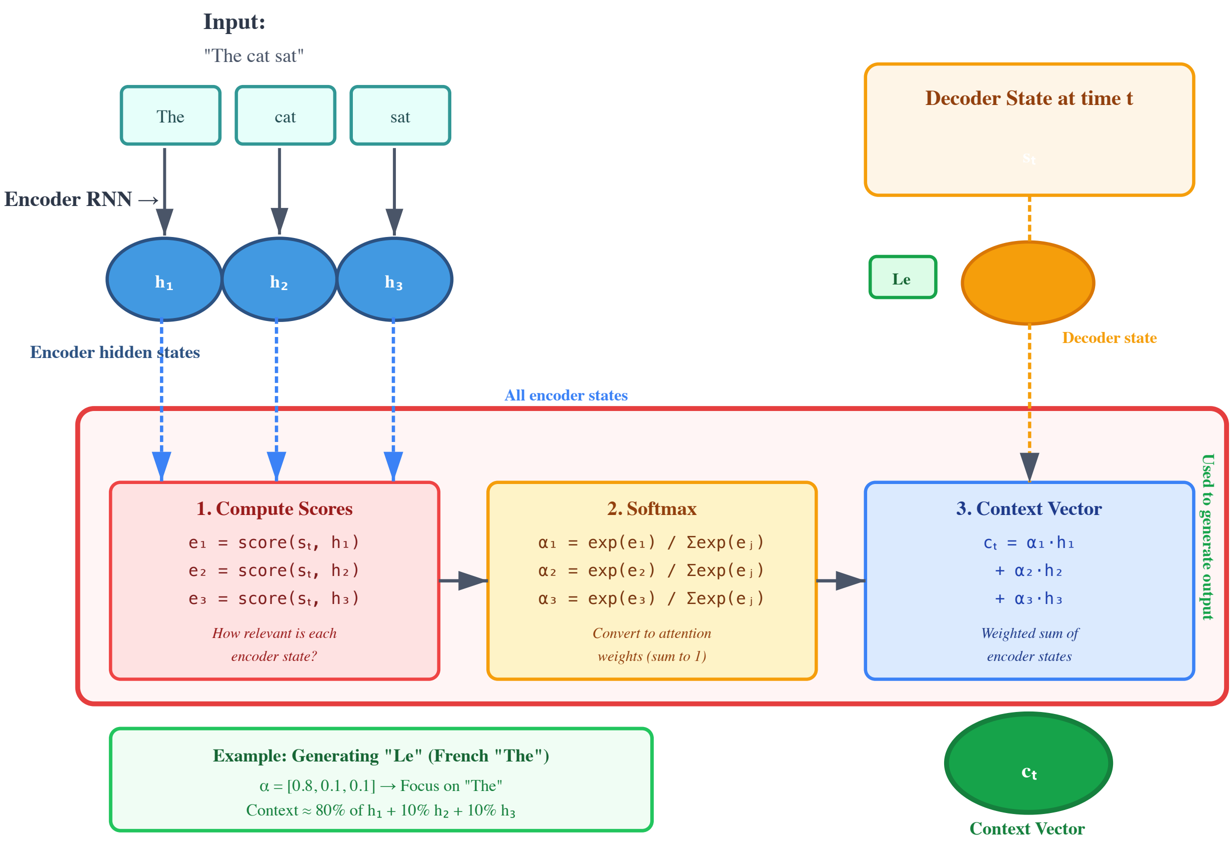
3. **Convert to attention weights** via softmax:

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{n} \exp(e_{tk})}$$

4. **Compute context vector** as weighted sum:

$$c_t = \sum_{j=1}^{n} \alpha_{tj} h_j$$

5. **Use c_t for decoding** : Generate output word based s_t on both decoder state and context
6. It is possible to build to series of hidden states: $h_1, h_2, \ldots, h_n$ (from first to last word) and $h_n, h_{n-1}, \ldots, h_1$ (from last to first) and combine them



Example in translation:

```
Translating: "The cat sat" → "Le chat s'est assis"

When generating "chat" (French for cat):
- Decoder computes attention over ["The", "cat", "sat"]
- High attention weight on "cat": α = [0.1, 0.8, 0.1]
- Context vector emphasizes "cat" representation
- Decoder uses this to generate correct translation
```

Model training

1. specify the encoder and decoder models (LSTM, GRU)
2. add attention module
3. build a translation module (sequence-to-sequence model too)
4. train all

Why It Mattered

Impact on the field:

- ✅ **Solved the bottleneck**: No need to compress entire sequence into single vector
- ✅ **Enabled long sequences**: Could handle 50+ word sentences effectively
- ✅ **Interpretable**: Attention weights showed which source words influenced each target word

- ✅ **State-of-the-art results**: Dramatically improved translation quality

Difference from Transformers:

- **Still used RNNs**: Attention was an *add-on* to RNN encoder-decoder
- **Sequential processing**: Had to process words one at a time (no parallelization)
- **Auxiliary mechanism**: Attention helped RNNs, but RNNs were still the core architecture

The path to Transformers:

```
Bahdanau Attention (2014): "Let's add attention TO RNNs"
                              ↓
                   Huge improvement!
                              ↓
Vaswani et al. (2017): "What if we use ONLY attention?"
                              ↓
                    Transformer is born
```

Mathematical Formulation

The Bahdanau attention mechanism can be written as:

$$context_t = \sum_{j=1}^{n} softmax(score(s_t, h_j)) \cdot hj$$

where the score function learns to measure
alignment between decoder state s_ts_t $standencoderstate\$h_j$

This is conceptually similar to the Transformer's attention:

$$Attention(Q, K, V) = softmax(QK^T)V$$

but

- Bahdanau: Single query (decoder state) attending to sequence
- Transformer: All positions attend to all positions simultaneously
- Bahdanau: Score computed via learned MLP
- Transformer: Score computed via dot product (simpler, faster)

Bahdanau attention
- proved that explicit attention mechanisms could dramatically improve sequence-to-sequence models.
- showed that models could "look back" at relevant input positions rather than relying on a compressed context
- inspired the Transformer, which took the idea further by making attention the *only* mechanism, removing RNNs entirely.

Transformer explained

Attention Intuition

```
Translation: "The cat sat on the mat" → "Le chat s'est assis sur le tapis"

When generating "chat":
Attention weights: [ 0.8, 0.15, 0.02, 0.01, 0.01, 0.01 ]
                    ↓     ↓     ↓     ↓     ↓     ↓
                   chat  s'est assis sur   le    tapis

Model focuses on "cat" (0.8 weight) while translating!
```

From Implicit to Explicit Attention

- **RNN Encoder-Decoder (2014):**
  - Implicit attention through recurrent connections
    - Each h*t depends on all previous*
      *$$h_t = f(h*{t-1}, x_t)$$
- **Bahdanau Attention (2015):**
  - Explicit attention weights

$$\alpha_t = softmax\left(\text{score}(h_t, \text{encoder-states})\right)$$
$$context_t = \sum \alpha_t[i] * \text{encoder-states}[i]$$

- **Self-Attention / Transformer (2017):**
  - Attention as the PRIMARY mechanism (not auxiliary)
    - No RNN at all

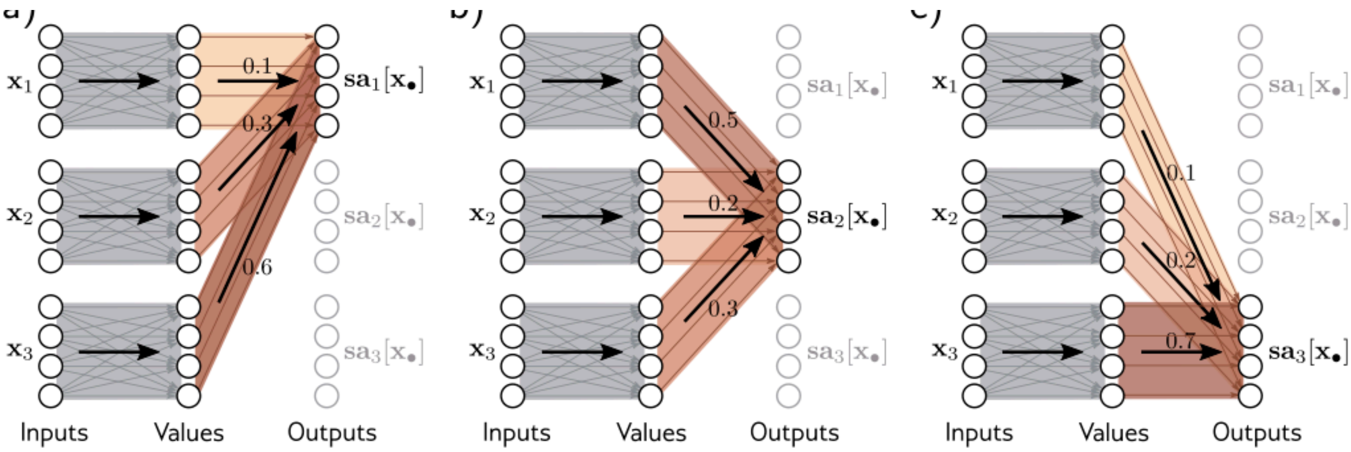
$$output = Attention(Q, K, V)$$

---

5. Attention Mechanism

- model needs to cope with input passages (e.g. in NLP) of different lengths
- know connections between elements that depend on attention
- transformer gets both by using **dot-product self-attention**

Self-Attention: The Core Operation



(from Understanding deep learning)

- standard network computes a linear transformation with a non-linear function
- **self-attention** $sa[]$ takes
  - N inputs
  - returns N output vectors of the same size (e.g., in NLP a word or some sub-word)
- a **value** is computed from each token

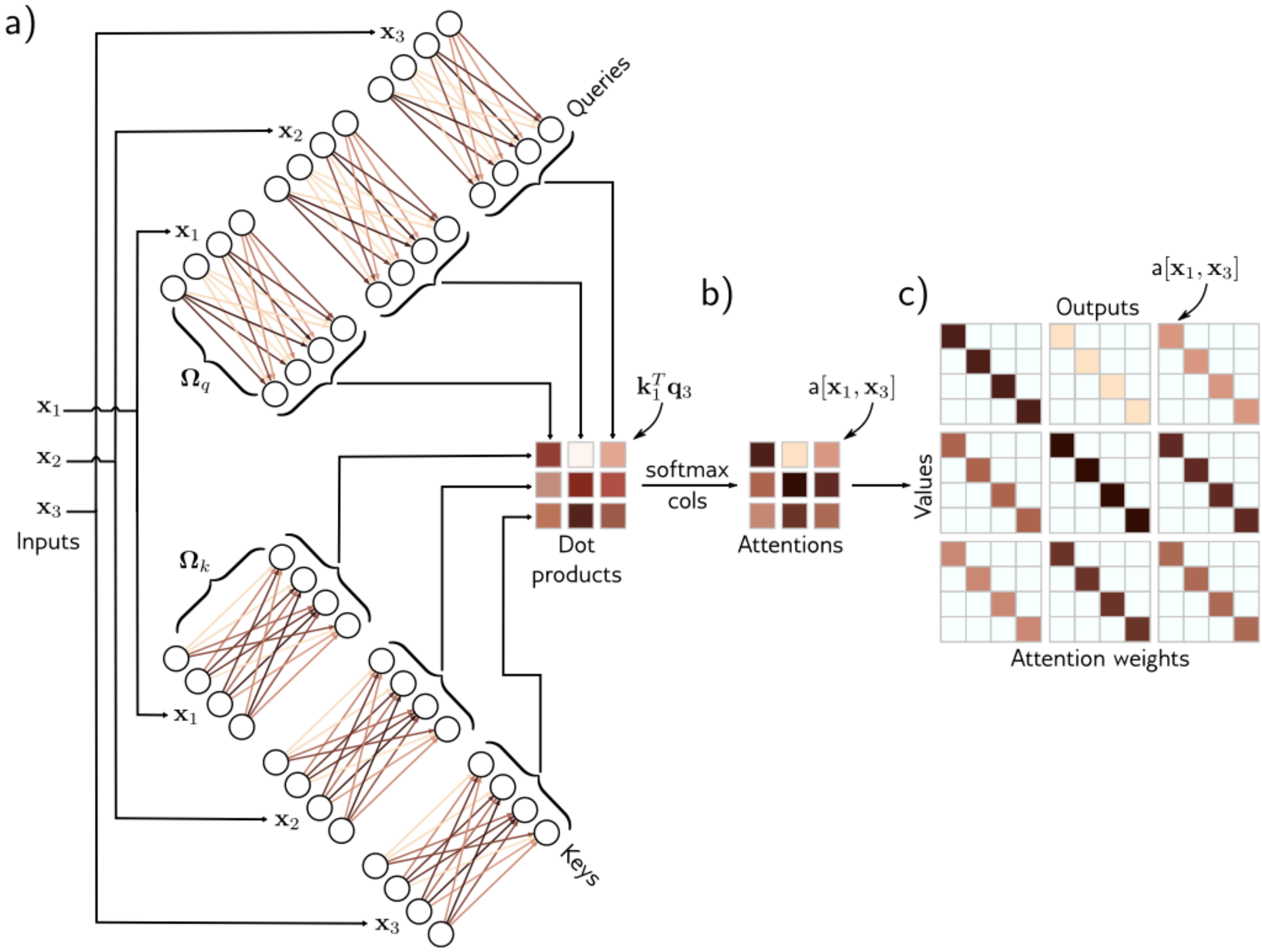
$$v_m = \beta_v + \Omega_v x_m$$

  using biases β and weights Ω
  - may be done in parallel
- a scalar $a[x_i, x_j]$ is the **attention** that token x_j pays to token x_i
  - attentions $a[\cdot, x_j]$ sum up to 1
- **self attention**

$$sa_j(x_1, \ldots, x_N) = \sum_{i=1}^{N} a[x_i, x_j] \cdot v_i$$

  - a weighted sum of all values v_i

    - weights $a(\cdot, x_n)$ are non-negative and sum-up to 1
    - each self-attention $sa_i[x_1, \ldots]$ can be thought as routing of the original N tokens with different proportions for the current task
- all can be computed in parallel per token
- in the figure above
    - N inputs are taken
    - in the left-most a routing to $sa_1(x)$ is computed with weights $0.1, 0.3, 0.6$
    - then two different **routings**



(from Prince, Understanding deep learning, MIT,2023)

- *query* vectors are computed as $q_n = \beta_q + \Omega_q x_n$
- *key* vectors are computed as $k_n = \beta_k + \Omega_q x_n$
- dot products are passed to a softmax giving attention values

Given input $\mathbf{X} \in \mathbb{R}^{N \times D}$:

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

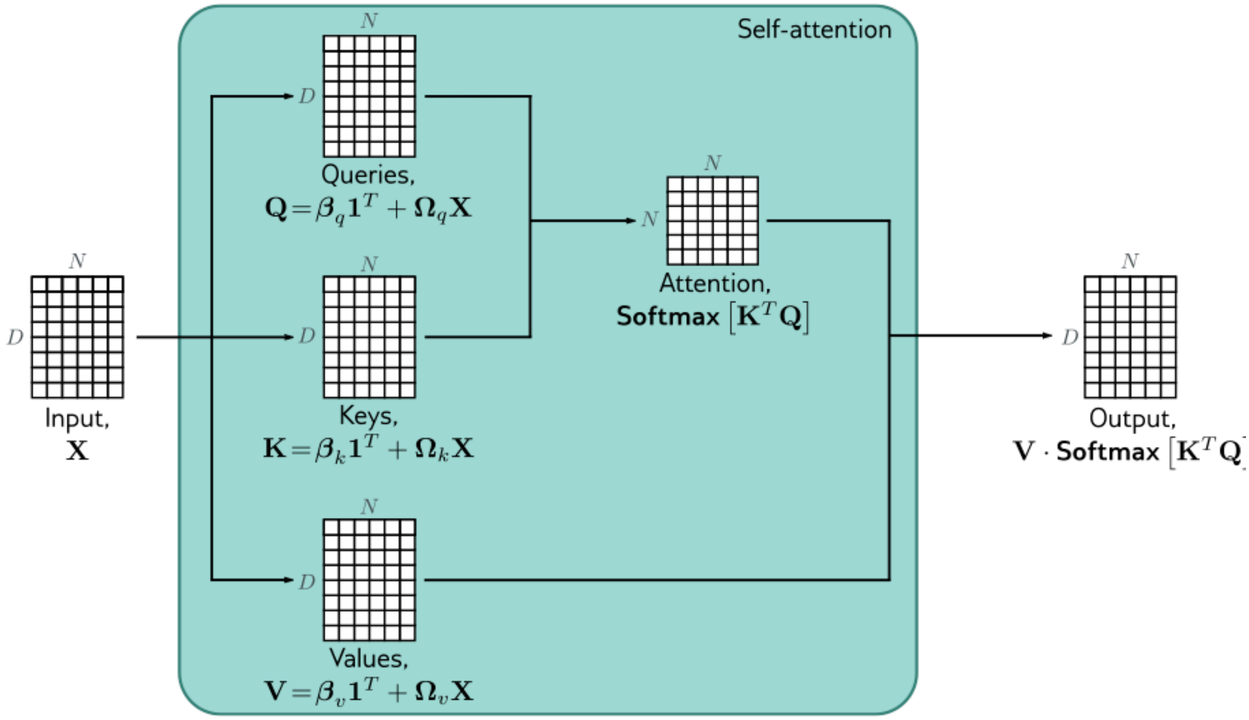
Where:

- $Q = XW_Q$ (Queries): "What am I looking for?"
- $K = XW_K$ (Keys): "What information do I have?"
- $V = XW_V$ (Values): "What information to aggregate?"
  where dimensions are
- d model dimension, size of the embedding,
- d_k the key and query dimension: $d_k = d/n_{heads}$ (standard relationship)
- $d_q = d_k$ (typically $d_q = d_k$)
- d_v dimension of values in attention: $d_v = d$
    - typically embedding dimension needs to be a multiple of the number of heads

- if $d_q \neq d_k$, an additional attention may be used where additional trained matrices map both to the same dimension
- $$W_Q \in \mathbb{R}^{d \times d_q}, W_K \in \mathbb{R}^{d \times d_k}, W_V \in \mathbb{R}^{d \times d_v}$$
- dot products in computation may get to large values
- and the to softmax regions where largest value dominates, and the gradients get very small
- model gets hard to train
- the scaling by the squere root of the number of rows of keys (and queries) prevents it

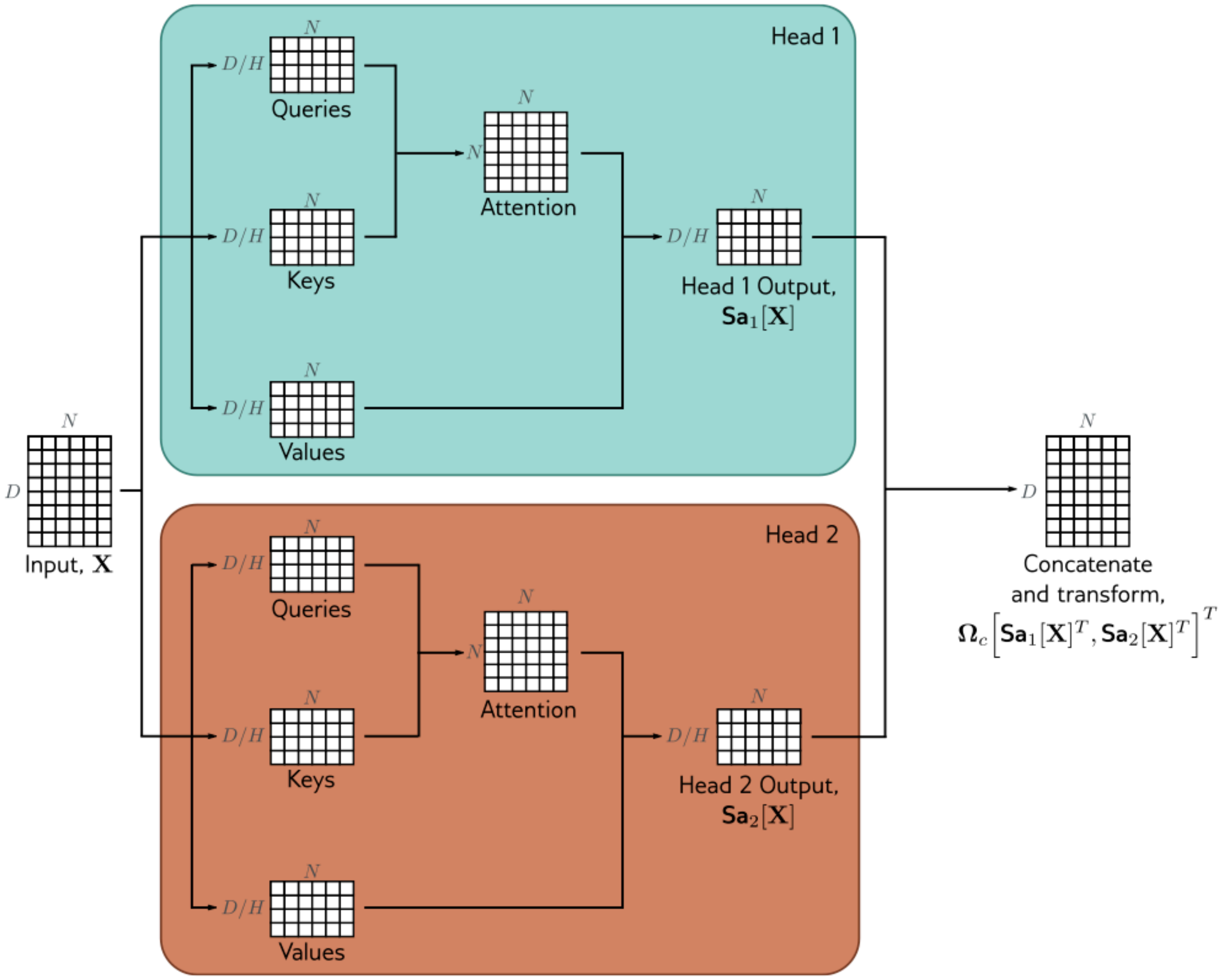
Key, query, value

- the **dot product** returns a measure of similarity between its arguments
- weights $a[x_., x_n]$ depend on relative similarities between the n-th query and all the keys
- queries and keys should have the same dimensions
  - there is a possibility of mapping to a common value



Multiple attention heads

- multiple heads may be computed in parallel



- typically for model dimension d and num_{heads} heads, the values, queries and keys will **all be of the same size** allowing for efficient computation
- multiple heads are concatenated

---

3. Understanding Query, Key, and Value: The Heart of Attention

Beyond the Matrices: Building Intuition

[Transformer explainer]

The Fundamental Question

Why do we need THREE separate matrices (Q, K, V)?

Bad answer: "Because the math works out"

- in my (/igor) opinion the [Attention Is All You Need, Vaswani et al., 2017)] paper caused Transformers a great deal of harm
- people now understand Transformer just as clever multiplication of matrices
- almost nobody asks themselves how the model "thinks", stores knowledge

Good answer: "Because attention is answering three distinct questions:"

1. **Query (Q)**: "What information am I looking for?"
2. **Key (K)**: "What information do I offer?"
3. **Value (V)**: "What is the actual information I provide?"

Library Search

```
You walk into a library looking for:
QUERY: "Books about neural networks written after 2017"
```

The librarian's process:

1. **Check catalog cards (KEYS)**:
   - Book 1: "Deep Learning, 2016, Goodfellow"
   - Book 2: "Neural Networks, 2020, Smith"
   - Book 3: "Transformers, 2017, Vaswani"
2. **Match your query against keys**:
   - Book 1: ❌ (too old)
   - Book 2: ✅✅ (perfect match!)
   - Book 3: ✅ (borderline - exactly 2017)
3. **Retrieve actual books (VALUES)**:
   - The catalog card (KEY) tells you WHERE to look
   - But what you get is the actual BOOK (VALUE)
   - Key = "metadata for matching"
   - Value = "the content you actually want"

In attention:

```
Your query: "Books about neural networks after 2017"
           ↓
Similarity scores: [0.1, 0.9, 0.5]  # How well each book matches
           ↓
Weighted retrieval: 0.1 * Book₁ + 0.9 * Book₂ + 0.5 * Book₃
           ↓
You get: Mostly Book 2, some Book 3, almost no Book 1
```

Database Query (SQL-like)

```sql
SELECT value_column
FROM table
WHERE key_column MATCHES query
ORDER BY similarity(key_column, query) DESC

-- In attention, this becomes:
-- Weighted retrieval based on soft matching!
```

Example: Student Database

```python
# You want information about: "CS students with GPA > 3.5"
query = [field: "CS", criteria: "GPA > 3.5"]

# Database entries:
students = [
    {key: "Alice, CS, 3.9",     value: "Research: NLP, Skills: PyTorch"},
    {key: "Bob, Math, 3.8",     value: "Research: Topology, Skills: Proofs"},
    {key: "Carol, CS, 3.2",     value: "Research: Graphics, Skills: OpenGL"},
    {key: "Dave, CS, 3.7",      value: "Research: ML, Skills: TensorFlow"}
]

# Attention scores (how well keys match query):
scores = [0.95, 0.1, 0.3, 0.9]  # Alice and Dave match best

# Weighted retrieval (soft SQL):
result = 0.95 * "Research: NLP..." + 0.1 * "Research: Topology..."
       + 0.3 * "Research: Graphics..." + 0.9 * "Research: ML..."
```

```
# You get: Mixture of Alice and Dave's info (CS students with high GPA)
```

Key insight:

- **Keys** are for matching/indexing
- **Values** are the actual content you retrieve
- **Query** is what you're searching for

Search Engine

```
User types: "best restaurants near me"  ← QUERY
                    ↓
Google's index:
┌─────────────────────────────────────────────┐
│ Document 1:                                 │
│   KEY: [location:nearby, topic:restaurants, │
│         rating:4.5, recency:2024]           │
│   VALUE: "Mario's Pizza: 123 Main St,       │
│           4.5★, Italian cuisine..."         │
└─────────────────────────────────────────────┘

┌─────────────────────────────────────────────┐
│ Document 2:                                 │
│   KEY: [location:nearby, topic:hotels,      │
│         rating:4.0, recency:2023]           │
│   VALUE: "Grand Hotel: 456 Oak Ave,         │
│           4.0★, Luxury accommodation..."    │
└─────────────────────────────────────────────┘
```

Matching process:

1. **Encode query**: "restaurants nearby" → query vector
2. **Compare to keys**: Which documents are relevant?
   - Doc 1: topic=restaurants ✓, location=nearby ✓ → HIGH score
   - Doc 2: topic=hotels ✗, location=nearby ✓ → LOW score
3. **Retrieve values**: Return actual document content (weighted by scores)

Why separate Key and Value?

- **Key**: Optimised for fast similarity search
  - Compressed representation: [location, topic, rating, recency]
  - Like a hash/index for quick lookup
- **Value**: The full, rich information
  - Complete document content
  - Much larger, more detailed
  - You don't search through this directly (too expensive!)

The Mathematical Picture: Information Retrieval

From Retrieval to Continuous Attention

Hard retrieval (traditional):

```python
def hard_lookup(query, keys, values):
    # Find BEST matching key
    best_idx = argmax(similarity(query, keys))
    # Return corresponding value
    return values[best_idx]

# Problem: Discrete, non-differentiable
```

Soft retrieval (attention):

```
def soft_lookup(query, keys, values):
    # Compute similarity to ALL keys
    scores = similarity(query, keys)  # [s₁, s₂, ..., sₙ]

    # Convert to probabilities (softmax)
    weights = softmax(scores)  # [w₁, w₂, ..., wₙ]

    # Weighted average of ALL values
    return Σ wᵢ * values[i]

# Differentiable! Can learn what to attend to
```

- most Transformer architectures use some form of top-k architecture
  - in NN top-k means applying the weights modification **only** to the highest **k** outputs
  - thus only the meaningful classes are modified
  - it can be done using differentiable operations, not just cutting off

Visualisation of scores → weights:

```
Raw scores:         [2.1,      0.3,     -0.5,      1.8]
                     ↓ softmax
Attention weights: [0.58,     0.10,     0.04,     0.28]  # Sum to 1.0
                     ↓
Weighted sum:        0.58*V₁ + 0.10*V₂ + 0.04*V₃ + 0.28*V₄
```

Translating "The cat sat on the mat" → French

Setup: Decoder generating "chat" (French for "cat")

Semantic dimensions:

1. **Is-Animal** (0-1)
2. **Is-Action** (0-1)
3. **Is-Object** (0-1)
4. **Definiteness** (0-1)

Step 1: The Query (from decoder)

Decoder's current state: "I'm trying to generate the French word for 'cat'"

```
Q = [0.9,  0.1,  0.2,  0.3]
     |     |     |     |
     |     |     |     └── definiteness (the cat, not just any cat)
     |     |     └────── is-object (0.2 - can be object but not here)
     |     └─────────── is-action (0.1 - not an action)
     └──────────────── is-animal (0.9 - STRONGLY looking for animal!)

# Query says: "I need something that is primarily an ANIMAL,
#              not an action, possibly definite"
```

Step 2: The Keys (from encoder - what each word "advertises")

Each English word broadcasts what it contains:

```
# "The" (first occurrence)
K_the₁ = [0.0,  0.0,  0.0,  1.0]
          |     |     |     └─ definiteness=1.0 (it's "the"!)
          └ is-animal=0.0 (not an animal)

# "cat"
```

```
K_cat  = [1.0,  0.0,  0.3,  0.0]
          │     │     │     └─ definiteness=0.0 (not a determiner)
          │     │     └─────── is-object=0.3 (can be object)
          │     └───────────── is-action=0.0 (not an action)
          └─────────────────── is-animal=1.0 (YES! I'm an animal!)

# "sat"
K_sat  = [0.0,  1.0,  0.0,  0.0]
          │     └─ is-action=1.0 (I'm a verb/action!)
          └─ is-animal=0.0 (not an animal)

# "on"
K_on   = [0.0,  0.0,  0.0,  0.0]
          └─ (preposition - low on all semantic features)

# "the" (second occurrence)
K_the₂ = [0.0,  0.0,  0.0,  1.0]
          └─ definiteness=1.0

# "mat"
K_mat  = [0.0,  0.0,  1.0,  0.0]
          │     │     └─────── is-object=1.0 (I'm an object!)
          │     └───────────── is-action=0.0
          └─────────────────── is-animal=0.0 (not an animal)
```

Step 3: Compute Attention Scores (Q · K)

```
Q = [is-animal, is-action, is-object, definitness]
```

Element-wise multiplication and sum:

```
# Q · K_the₁
score_the₁ = 0.9×0.0 + 0.1×0.0 + 0.2×0.0 + 0.3×1.0 = 0.3
            └──────────── no animal match  └─ some definiteness

# Q · K_cat
score_cat  = 0.9×1.0 + 0.1×0.0 + 0.2×0.3 + 0.3×0.0 = 0.96
            └─ STRONG animal match └─ minor object match

# Q · K_sat
score_sat  = 0.9×0.0 + 0.1×1.0 + 0.2×0.0 + 0.3×0.0 = 0.1
                      └─ action doesn't match our query

# Q · K_on
score_on   = 0.9×0.0 + 0.1×0.0 + 0.2×0.0 + 0.3×0.0 = 0.0
                                                 └─── preposition, no match

# Q · K_the₂
score_the₂ = 0.9×0.0 + 0.1×0.0 + 0.2×0.0 + 0.3×1.0 = 0.3
                                                 └─ same as first "the"

# Q · K_mat
score_mat  = 0.9×0.0 + 0.1×0.0 + 0.2×1.0 + 0.3×0.0 = 0.2
                                └─ some object match
```

Raw scores: [0.30, 0.96, 0.10, 0.00, 0.30, 0.20]

```
                 the₁   cat    sat    on     the₂   mat
                [0.30   0.96   0.10   0.00   0.30   0.20]
```

Step 4: Scale by $\sqrt{d_k}$

```
d_k = 4  # dimension of keys; needed for scalability
scaling_factor = √4 = 2.0

scaled_scores = [0.30, 0.96, 0.10, 0.00, 0.30, 0.20] / 2.0
              = [0.15, 0.48, 0.05, 0.00, 0.15, 0.10]
```

Step 5: Apply Softmax → Attention Weights

```
# exp(scaled_scores)
exp_scores = [exp(0.15), exp(0.48), exp(0.05), exp(0.00), exp(0.15), exp(0.10)]
           = [    1.16,       1.62,       1.05,       1.00,       1.16,       1.11 ]

# Normalize (sum = 7.10)
attention_weights = [1.16,  1.62,  1.05,  1.00,  1.16,  1.11] / 7.10
                  = [0.16,  0.23,  0.15,  0.14,  0.16,  0.16]
                α ≈ [0.16,    0.23,    0.15,    0.14,    0.16,    0.16]
                     the₁     CAT      sat      on       the₂     mat
```

Visualization:

```
Word:   the₁   cat   sat   on    the₂   mat
Weight: 16%    23%   15%   14%   16%    16%

        ▬▬▬    ▬▬▬   ▬▬    ▬▬    ▬▬▬    ▬▬▬
                ↑
           Highest attention to "cat"!
```

Step 6: The Values (actual semantic content to retrieve)

Values contain rich, contextualized information:

```
# V_the₁: Determiner introducing "cat"
V_the₁ = [0.1,    0.0,    0.0,    0.8]
          │                        └─ definite article marker
          └─ minimal semantic content

# V_cat: Rich animal semantics
V_cat  = [0.9,    0.0,    0.1,    0.7]
          │       │       │       └─ definite (preceded by "the")
          │       │       └─────── minor object role
          │       └───────────── not action
          └──────────────── ANIMAL (feline, domestic, pet)

# V_sat: Action/state information
V_sat  = [0.0,    0.9,    0.0,    0.3]
          │       └─ PAST ACTION (sitting)
          └─ not animal

# V_on: Prepositional relationship
V_on   = [0.0,    0.0,    0.5,    0.0]
          │               └─ spatial relationship
          └─ not animal

# V_the₂: Determiner for "mat"
V_the₂ = [0.0,    0.0,    0.0,    0.9]
          └─────────── definite article

# V_mat: Object semantics
V_mat  = [0.0,    0.0,    0.9,    0.6]
          │               │       └─ definite object
          │               └─────── OBJECT (floor covering)
          └──────────────── not animal
```

Step 7: Weighted Sum of Values (Final Output)

```
output = α₁·V_the₁ + α₂·V_cat + α₃·V_sat + α₄·V_on + α₅·V_the₂ + α₆·V_mat

       = 0.16·[0.1,  0.0,  0.0,  0.8] + 0.23·[0.9,  0.0,  0.1,  0.7]
       + 0.15·[0.0,  0.9,  0.0,  0.3] + 0.14·[0.0,  0.0,  0.5,  0.0]
       + 0.16·[0.0,  0.0,  0.0,  0.9] + 0.16·[0.0,  0.0,  0.9,  0.6]

# Dimension 1 (is-animal):
= 0.16×0.1 + 0.23×0.9 + 0.15×0.0 + 0.14×0.0 + 0.16×0.0 + 0.16×0.0
= 0.016 + 0.207 + 0 + 0 + 0 + 0
= 0.223

# Dimension 2 (is-action):
= 0.16×0.0 + 0.23×0.0 + 0.15×0.9 + 0.14×0.0 + 0.16×0.0 + 0.16×0.0
= 0 + 0 + 0.135 + 0 + 0 + 0
= 0.135

# Dimension 3 (is-object):
= 0.16×0.0 + 0.23×0.1 + 0.15×0.0 + 0.14×0.5 + 0.16×0.0 + 0.16×0.9
= 0 + 0.023 + 0 + 0.070 + 0 + 0.144
= 0.237

# Dimension 4 (definiteness):
= 0.16×0.8 + 0.23×0.7 + 0.15×0.3 + 0.14×0.0 + 0.16×0.9 + 0.16×0.6
= 0.128 + 0.161 + 0.045 + 0 + 0.144 + 0.096
= 0.574

output = [0.223,    0.135,    0.237,    0.574]
          │         │         │         └─ Strong definiteness signal
          │         │         └─────── Some object context
          │         └─────────── Some action context
          └─────────────── Animal signal (from "cat")
```

Interpretation of the Output

The enriched representation `[0.22, 0.14, 0.24, 0.57]` means:

```
┌─────────────────────────────────────────────────┐
│ Original query was looking for: ANIMAL          │
│                                                 │
│ Got back:                                       │
│   • 22% animal features     ← from "cat" (23%)  │
│   • 14% action features     ← from "sat" (15%)  │
│   • 24% object features     ← from "mat" (16%)  │
│   • 57% definiteness        ← from "the"s       │
│                                                 │
│ Decoder now knows:                              │
│   "Generate an animal word (cat → chat),        │
│    it's definite (le/la),                       │
│    with some action/object context"             │
└─────────────────────────────────────────────────┘
```

Why this helps translation:

- Primary signal: ANIMAL (0.22) → Generate "chat" (cat)
- Definiteness (0.57) → Use "le chat" not just "chat"
- Context from action/object → Past tense, spatial relationship

---

Visual Summary: The Complete Flow

```
INPUT: "The cat sat on the mat"
          ↓
ENCODER REPRESENTATIONS:
```

```
┌─────────────────────────────────────────────────────────┐
│  Word   │   Key (for matching)   │   Value (content)     │
├─────────┼────────────────────────┼───────────────────────┤
│  the₁   │ [0.0, 0.0, 0.0, 1.0]   │ [0.1,  0.0,  0.0,  0.8] │
│  cat    │ [1.0, 0.0, 0.3, 0.0]   │ [0.9,  0.0,  0.1,  0.7] │ ←┐
│  sat    │ [0.0, 1.0, 0.0, 0.0]   │ [0.0,  0.9,  0.0,  0.3] │  │
│  on     │ [0.0, 0.0, 0.0, 0.0]   │ [0.0,  0.0,  0.5,  0.0] │  │
│  the₂   │ [0.0, 0.0, 0.0, 1.0]   │ [0.0,  0.0,  0.0,  0.9] │  │
│  mat    │ [0.0, 0.0, 1.0, 0.0]   │ [0.0,  0.0,  0.9,  0.6] │  │
└─────────────────────────────────────────────────────────┘  │
          ↓                                                   │
DECODER QUERY: [0.9,  0.1,  0.2,  0.3]                        │
    "Looking for an animal"                                   │
          ↓                                                   │
ATTENTION SCORES (Q·K):                                       │
    [0.30,  0.96,  0.10,  0.00,  0.30,  0.20]                 │
            ↑↑↑ Highest match!                                │
            ↓                                                 │
ATTENTION WEIGHTS (softmax):                                  │
    [0.16,  0.23,  0.15,  0.14,  0.16,  0.16]                 │
            ↑↑↑ Focus on "cat"                                │
            ↓                                                 │
OUTPUT (weighted sum of Values): ─────────────────────────────┘
    [0.22,  0.14,  0.24,  0.57]
     └─ Mostly animal features, definite, some context
          ↓
DECODER GENERATES: "chat"
```

---

1. Keys are for matching

- `K_cat = [1.0, 0.0, 0.3, 0.0]` says "I'm an animal"
- Query `Q = [0.9, 0.1, 0.2, 0.3]` matches strongly (score = 0.96)
- Other keys don't match as well

2. Values are for content

- Even though we matched on "cat" via Key
- We retrieve rich Value: `[0.9, 0.0, 0.1, 0.7]`
- Contains more than just "is-animal" flag
- Has definiteness, context, semantic richness

3. Soft retrieval averages context

- Not just "cat" (23% weight)
- Also gets definiteness from "the" (16% + 16%)
- Some action context from "sat" (15%)
- Some object context from "mat" (16%)

4. Different queries would give different results

- If query was `[0.1, 0.9, 0.1, 0.1]` (looking for action)
  - Would attend to "sat" instead
  - Would retrieve action semantics
  - Would generate French verb

Why Not Just Use One Matrix?

Thought Experiment: What if Q = K = V?

```
# Self-attention with Q = K = V = X
attention = softmax(X @ X.T) @ X
```

```
# This means:
# - Matching criterion = Content itself
# - Retrieved content = Same as matching criterion
```

Problem 1: Conflation of "what to match" and "what to retrieve"

Example from translation:

```
Word "bank" in: "I went to the bank by the river"

If Key = Value:
- Key says: "I'm the word 'bank' (financial OR river-side)"
- Value says: Same information

But we want:
- Key to say: "I'm a noun that relates to 'river'"
  → Helps matching in this context
- Value to provide: Full semantic content for "river bank"
  → Different from just the word form
```

Problem 2: Reduced expressiveness

```
# With Q = K = V (1 matrix):
Parameters: d²

# With Q, K, V separate (3 matrices):
Parameters: 3d²

# More parameters = more expressive power
# Can learn richer representations
```

Problem 3: Asymmetry of roles

```
Query:  "What do I need?" (generated by current state)
Key:    "What can I offer?" (advertised by candidates)
Value:  "Here's what I actually give" (delivered content)

These are fundamentally different questions!
Forcing them to be the same limits what the model can learn.
```

The Key-Value Separation: A Deeper Dive

Why is Key ≠ Value Important?

Intuition: Index vs Content

Think of a book:

- **Key** = Chapter titles, keywords, page numbers (the INDEX)
  - Optimized for: Quick scanning, pattern matching
  - Characteristics: Compressed, abstract, matchable
- **Value** = Actual chapter content (the TEXT)
  - Optimized for: Rich information, semantic content
  - Characteristics: Detailed, specific, informative

```
# Book: "Deep Learning"
key_chapter3 = "Chapter 3: Linear Algebra Review"
              # Concise, tells you what's inside

value_chapter3 = """
    Linear algebra is the study of vectors, matrices...
    [20 pages of detailed content]
```

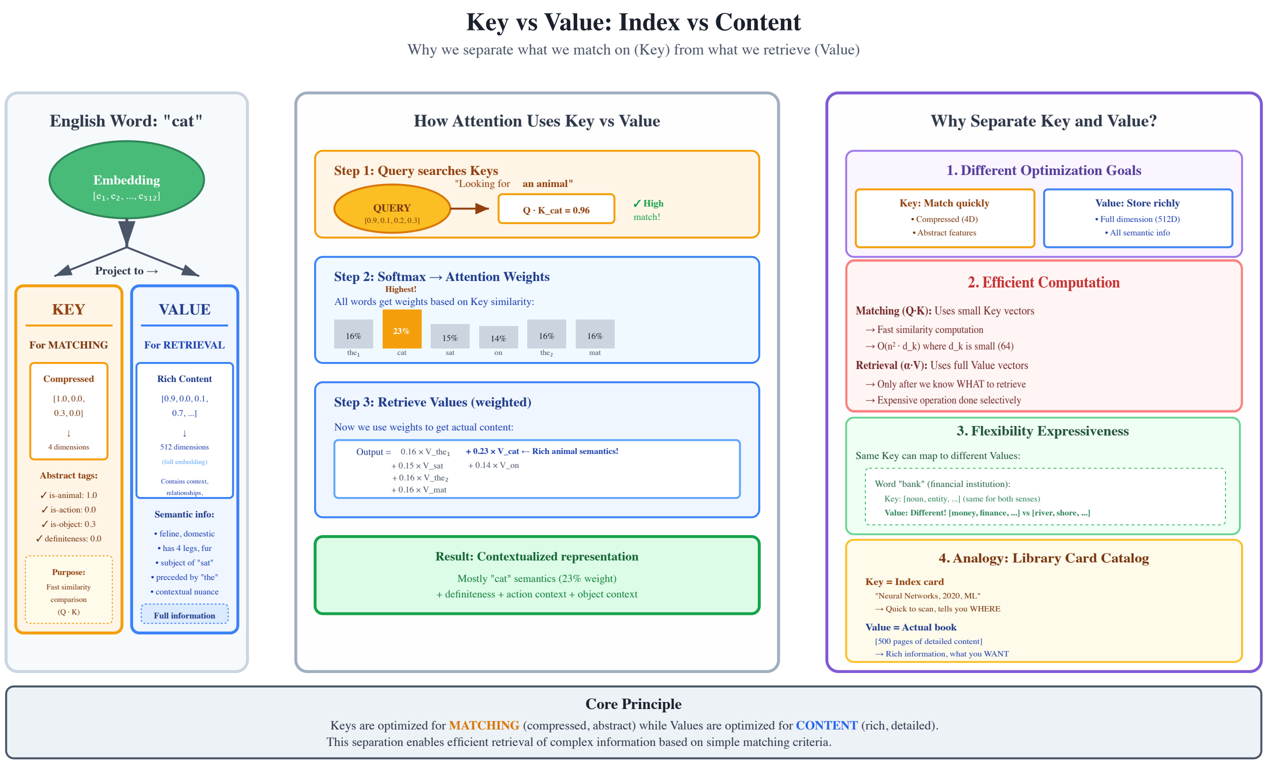
```
"""
              # Rich, detailed information
```

In neural networks:

```
# Encoding a sentence: "The black cat"

# Keys: Compressed, abstract representations for matching
K_the   = embed("determiner, definite")
K_black = embed("adjective, color, descriptor")
K_cat   = embed("noun, animal, subject")

# Values: Rich semantic embeddings with full context
V_the   = embed("article with context from 'black cat'")
V_black = embed("color=black, modifies=cat, visual_property")
V_cat   = embed("animal=feline, color=black, subject=yes, ...")
         # Much richer! Includes compositional information
```



---

4. The Attention Mechanism: Mathematics

Core Attention Equation

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

Where:

- Q (Query): "What am I looking for?" — $\mathbb{R}^{n \times d_k}$ (no of queries $\times$ key / query dimension)
- K (Key): "What do I contain?" — $\mathbb{R}^{m \times d_k}$ (no of keys / values $\times$ key / query dimension)
- V (Value): "What information do I have?" — $\mathbb{R}^{m \times d_v}$ (no of keys / values $\times$ value dimension)

Dimensions:

- n: Number of queries (target sequence length)
- m: Number of keys/values (source sequence length)
- d_k: Key/query dimension (for scalability - see below)
- d_v: Value dimension

Attention as Soft Dictionary Lookup

Analogy:

```
# Traditional dictionary
dictionary = {
    "cat": "a small domesticated carnivorous mammal",
    "dog": "a domesticated carnivorous mammal",
    ...
}
result = dictionary["cat"]  # Hard lookup: exact match

# Attention as soft dictionary
queries = ["ct", "catt", "dog"]  # Fuzzy queries
keys = ["cat", "dog", "bird"]
values = [embedding_cat, embedding_dog, embedding_bird]

# Each query attends to ALL keys, weighted by similarity
result = Σ similarity(query, key_i) * value_i
```

Step-by-Step Computation

Given input sequence $X = [x_1, x_2, \ldots, x_n] \in \mathbb{R}^{n \times d}$:

Step 1: Linear Projections

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

where $W^Q, W^K \in \mathbb{R}^{d \times d_k}$ and $W^V \in \mathbb{R}^{d \times d_v}$

Step 2: Compute Attention Scores

$$\text{scores} = QK^T \in \mathbb{R}^{n \times n}$$

Element (i, j) measures how much query i should attend to key j

Step 3: Scale (Important!)

$$\text{scaled scores} = \frac{QK^T}{\sqrt{d_k}}$$

Why divide by $\sqrt{d_k}$?

$$q \cdot k = \sum_i^d q_i k_i$$

- q_i and k_i have 0 mean and 1 variance
- $q_i k_i$ has variance 1 too
- the sum's variance is d and standard deviation $\sqrt{d}$
- if scaled $q \cdot k / \sqrt{d}$, the whole variance is 1 too regardless of dimension
- division keeps gradients stable (prevents softmax saturation)

Step 4: Apply Softmax

$$\alpha = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

Each row sums to 1: $\sum_j \alpha_{ij} = 1$

Step 5: Weighted Sum of Values

$$\text{Output} = \alpha V$$

Attention Score Interpretation

$$\text{score}_{ij} = q_i \cdot k_j = |q_i||k_j|\cos(\theta_{ij})$$

- High score → query i and key j are "aligned"
- Low score → query i doesn't need key j

After softmax:

$$\alpha_{ij} = \frac{\exp(\text{score}_{ij})}{\sum_k \exp(\text{score}_{ik})}$$

Attention weights form a probability distribution over source positions

---

Self-Attention vs Cross-Attention

Type	Q from	K,V from	Use Case
Self-Attention	Same sequence	Same sequence	Encoding contextual relationships
Cross-Attention	Target sequence	Source sequence	Encoder-Decoder connection

Self-Attention:

```
# All positions attend to all other positions in same sequence
Q = K = V = X  # Same source
# Learns internal structure and dependencies
```

Cross-Attention:

```
# Decoder attends to encoder
Q = decoder_states  # What decoder is looking for
K = V = encoder_states  # Information from encoder
# Connects source and target sequences
```

---

5. Multi-Head Attention

The Motivation

Single attention head:

- Learns one notion of "relatedness"
- May miss different types of relationships

Example:

```
Sentence: "The bank by the river is steep"

Head 1 might learn: syntactic relationships (subject-verb)
Head 2 might learn: semantic relationships (bank-river)
Head 3 might learn: positional relationships (nearby words)
```

Multi-Head Attention Formula

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

where each head:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

where

- h: number of heads (typically 8 or 16)
- $W_i^Q, W_i^K \in \mathbb{R}^{d_{model} \times d_k}$ where $d_k = d_{model}/h$
- $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ where $d_v = d_{model}/h$
- $W^O \in \mathbb{R}^{hd_v \times d_{model}}$

Why Multiple Heads?

Mathematical Intuition:

Instead of one d_{model}-dimensional attention:

$$\text{Attention}(Q, K, V) \in \mathbb{R}^{n \times d_{model}}$$

Split into h heads of dimension $d_k = d_{model}/h$:

$$\text{head}_i \in \mathbb{R}^{n \times d_k}$$

1. **Subspace specialization**: Each head can attend to different aspects
2. **Parameter efficiency**: h small projections vs 1 large projection
3. **Ensemble effect**: Multiple attention patterns averaged

Different attention patterns across heads [Transformer explained]

Computational View

```
# Pseudocode for Multi-Head Attention

def multi_head_attention(Q, K, V, num_heads=8):
    d_k = d_model // num_heads

    # Split into heads
    Q_heads = split_heads(Q, num_heads)  # (batch, heads, seq, d_k)
    K_heads = split_heads(K, num_heads)
    V_heads = split_heads(V, num_heads)

    # Parallel attention for each head
    attention_outputs = []
    for i in range(num_heads):
        head_i = scaled_dot_product_attention(
            Q_heads[:, i], K_heads[:, i], V_heads[:, i]
        )
        attention_outputs.append(head_i)

    # Concatenate and project
    concat = concatenate(attention_outputs)  # (batch, seq, d_model)
    output = linear(concat, W_O)

    return output
```

[Multihead attention in PyTorch]

Additional attention

If queries and keys do not have the same dimension, we may use an **additional** attention

$$attn(q, k) = w_v \tanh(W_q q + W_k k)$$

---

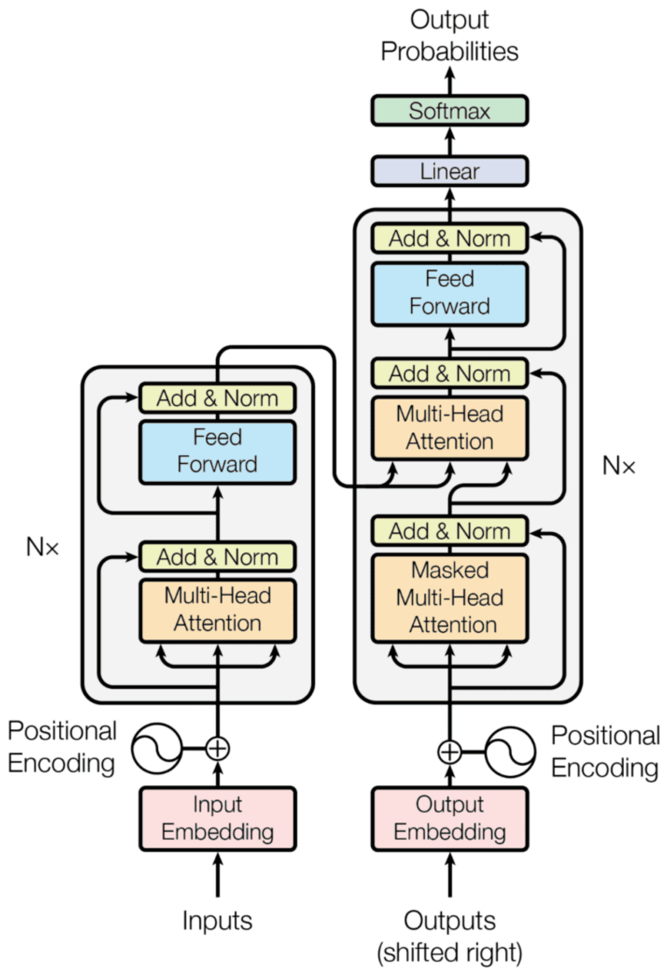
6. The Transformer Architecture

Three types of transformer architecture

- **encoder** transforms the embeddings into some representation that would supportsome processing task
- **decoder** predicts the next token to continue the input
- **encoder-decoder** for conversion of one sequence into another e.g., translation

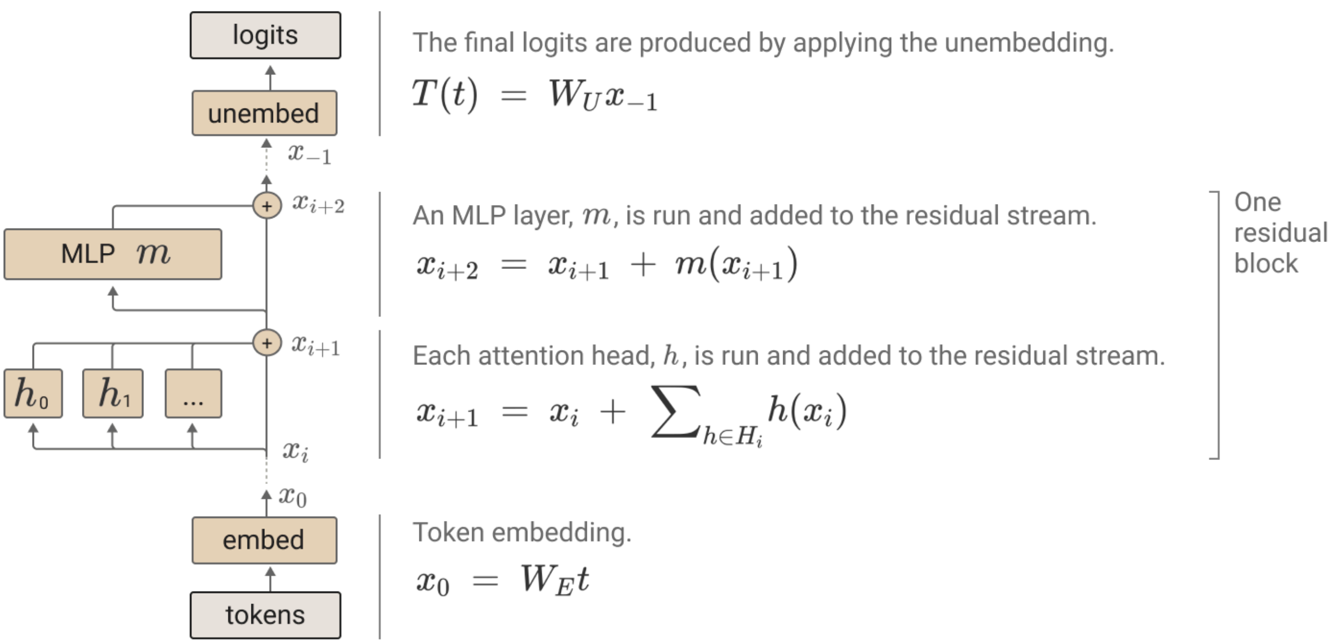
High-Level View

```
Input Sequence
      ↓
  [Embedding + Positional Encoding]
      ↓
  Encoder (× N layers)
  ┌─────────────────┐
  │ Multi-Head Attn │  (Self-attention)
  │       ↓         │
  │   Add & Norm    │
  │       ↓         │
  │  Feed-Forward   │
  │       ↓         │
  │   Add & Norm    │
  └─────────────────┘
      ↓
  Decoder (×N layers)
  ┌─────────────────┐
  │ Masked Attn     │  (Self-attention)
  │       ↓         │
  │   Add & Norm    │
  │       ↓         │
  │ Cross Attn      │  (Encoder-Decoder)
  │       ↓         │
  │   Add & Norm    │
  │       ↓         │
  │  Feed-Forward   │
  │       ↓         │
  │   Add & Norm    │
  └─────────────────┘
      ↓
  Linear + Softmax
      ↓
  Output Sequence
```



(after Vaswani et al.)
Full Transformer encoder (left) and decoder (right) architecture (one layer both)

- masked multi-head attention masks future tokens preventing glimpses "to the future"



(after Anthropic blog)

Transformer Encoder Layer

$$\text{EncoderLayer}(X) = \text{FFN}(\text{LayerNorm}(X + \text{MultiHead}(X, X, X)))$$

Detailed Steps:

1. **Self-Attention with Residual:**

$$Z = X + \text{MultiHead}(X, X, X)$$

2. **Layer Normalization:**

$$Z' = \text{LayerNorm}(Z)$$

3. **Feed-Forward with Residual:**

$$H = Z' + \text{FFN}(Z')$$

4. **Layer Normalisation:**

$$\text{Output} = \text{LayerNorm}(H)$$

Feed-Forward Network (FFN)

Architecture:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)\, W_2 + b_2$$

Properties:

- Applied **position-wise** (same FFN for each position independently)
- Two linear transformations with ReLU activation
- Typical dimensions: $d_{model} = 512$, $d_{ff} = 2048$

Why FFN after Attention?

1. **Non-linearity**: Attention is linear operations + softmax
2. **Mixing information**: Attention aggregates, FFN processes
3. **Capacity**: Adds parameters for complex transformations

FFN as 1×1 Convolution

The Position-Wise Feed-Forward Network

Standard FFN Definition

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

where:

- $x \in \mathbb{R}^{d_{model}}$ is a single position's representation
- $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}$, typically $d_{ff} = 4 \cdot d_{model}$
- $W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$ squeezes back to d_{model}

Key property: Applied **independently** to each position in the processed sequence

---

Applied to Full Sequence

Given sequence $X \in \mathbb{R}^{n \times d_{model}}$ where n is sequence length:

$$\text{FFN}(X) = \begin{bmatrix} \text{FFN}(x_1) \\ \text{FFN}(x_2) \\ \vdots \\ \text{FFN}(x_n) \end{bmatrix}$$

Crucial : Same weights W_1, W_2 applied to every position!

Equivalence to 1×1 Convolution

Reshape Perspective

Reshape sequence as "image": $X \in \mathbb{R}^{n \times 1 \times d_{model}}$

- n = "height" (sequence length)
- 1 = "width" (single position)
- d_{model} = "channels"

1×1 Convolution Operation

A 1×1 convolution with kernel $W \in \mathbb{R}^{1 \times 1 \times d_{in} \times d_{out}}$:

$$y_{i,j} = \sum_{c=1}^{d_{in}} W_{c,k} \cdot x_{i,j,c} + b_k$$

For each output channel k, at each spatial position (i, j)

The Mathematical Equivalence

FFN Layer 1: Linear + ReLU

$$Z = \max(0, XW_1 + b_1)$$

As 1×1 conv:

```
Conv1D(kernel_size=1, in_channels=d_model, out_channels=d_ff)
```

Dimension tracking:

```
Input:  (batch, seq_len, d_model)
         ↓
         ↓ reshape for conv
         ↓
        (batch, d_model, seq_len)  # channels first
         ↓
         ↓ 1×1 conv
         ↓
        (batch, d_ff, seq_len)
```

For position i:

$$z_i = \text{ReLU}\left(\sum_{c=1}^{d_{model}} W_1[c,:] \cdot x_i[c] + b_1\right)$$

This is exactly: **1×1 conv across channel dimension**

FFN Layer 2: Linear

$$\text{Output} = ZW_2 + b_2$$

As 1×1 conv:

```
Conv1D(kernel_size=1, in_channels=d_ff, out_channels=d_model)
```

Complete FFN as Two 1×1 Convolutions

$$\mathrm{FFN}(X) = \mathrm{Conv}_{1\times1}^{(2)}\left(\mathrm{ReLU}\left(\mathrm{Conv}_{1\times1}^{(1)}(X)\right)\right)$$

Explicit Form

$$\boxed{\begin{aligned}\mathrm{FFN}(X) &= \max(0, XW_1 + b_1)W_2 + b_2 \\ &\equiv \mathrm{Conv1D}_{1\times1}(\mathrm{ReLU}(\mathrm{Conv1D}_{1\times1}(X)))\end{aligned}}$$

PyTorch: Both Implementations are Identical

```
import torch
import torch.nn as nn

# Configuration
batch_size, seq_len, d_model, d_ff = 2, 10, 512, 2048

# ============================================
# Implementation 1: Standard FFN (Linear layers)
# ============================================
class FFN_Linear(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()

    def forward(self, x):
        # x: (batch, seq_len, d_model)
        x = self.linear1(x)      # (batch, seq_len, d_ff)
        x = self.relu(x)
        x = self.linear2(x)      # (batch, seq_len, d_model)
        return x

# ============================================
# Implementation 2: 1×1 Convolution
# ============================================
class FFN_Conv(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()
        # 1×1 convolution = kernel_size=1
        self.conv1 = nn.Conv1d(d_model, d_ff, kernel_size=1)
        self.conv2 = nn.Conv1d(d_ff, d_model, kernel_size=1)
        self.relu = nn.ReLU()

    def forward(self, x):
        # x: (batch, seq_len, d_model)

        # Conv1d expects (batch, channels, length)
        x = x.transpose(1, 2)    # (batch, d_model, seq_len)

        x = self.conv1(x)        # (batch, d_ff, seq_len)
        x = self.relu(x)
        x = self.conv2(x)        # (batch, d_model, seq_len)

        x = x.transpose(1, 2)    # (batch, seq_len, d_model)
        return x

# ============================================
# Verify they're equivalent
# ============================================
ffn_linear = FFN_Linear(d_model, d_ff)
ffn_conv = FFN_Conv(d_model, d_ff)

# Copy weights from linear to conv
ffn_conv.conv1.weight.data = ffn_linear.linear1.weight.data.unsqueeze(2)
ffn_conv.conv1.bias.data = ffn_linear.linear1.bias.data
ffn_conv.conv2.weight.data = ffn_linear.linear2.weight.data.unsqueeze(2)
ffn_conv.conv2.bias.data = ffn_linear.linear2.bias.data
```

```python
# Test on same input
x = torch.randn(batch_size, seq_len, d_model)

out_linear = ffn_linear(x)
out_conv = ffn_conv(x)

print(f"Outputs are equal: {torch.allclose(out_linear, out_conv, atol=1e-6)}")
# Output: Outputs are equal: True

print(f"Max difference: {(out_linear - out_conv).abs().max().item()}")
# Output: Max difference: ~1e-7 (numerical precision)
```

Why This Perspective Matters

1. Computational Interpretation

Position-wise = No mixing across positions

```python
# FFN does NOT look at neighboring positions
output[i] = f(input[i])  # Only depends on position i

# Unlike attention which mixes:
output[i] = Σⱼ attention[i, j] * input[j]  # Depends on all j
```

2. Comparison with CNNs

Operation	Receptive Field	Cross-position Mixing
1×1 Conv (FFN)	Single position	❌ No
3×3 Conv	3 positions	✓ Yes (local)
Attention	All positions	✓ Yes (global)

Insight:

- Attention = Global mixing (routes information)
- FFN = Local processing (transforms at each position)
- Together = Route then compute

---

3. Parameter Sharing

$$\text{Parameters} = d_{model} \times d_{ff} + d_{ff} \times d_{model}$$

Not $n \times d_{model} \times d_{ff}$ where n is sequence length!

Same weights applied to all positions → **Translation equivariance**

```python
# Shift the sequence by 1 position
X_shifted = torch.roll(X, shifts=1, dims=1)

# FFN output is also shifted by 1
assert torch.allclose(
    FFN(X_shifted),
    torch.roll(FFN(X), shifts=1, dims=1)
)
```

---

4. Why Not Use Larger Kernels?

Could use 3×3, 5×5 convolutions:

```
# This would mix neighboring positions
conv = nn.Conv1d(d_model, d_ff, kernel_size=3)
```

But the Transformer philosophy is to

- Attention handles cross-position mixing (global, content-dependent)
- FFN handles position-wise transformation (local, position-independent)
- Clean separation of concerns!

Mathematical Equivalence

$$\boxed{\text{FFN}_{\text{position-wise}} \equiv \text{Conv1D}_{\text{kernel=1}}}$$

1. **Conceptual clarity**: FFN doesn't mix positions, only transforms at each position
2. **Computational efficiency**: Can use optimised conv implementations
3. **Framework understanding**: Attention = mixing, FFN = processing
4. **Architecture design**: Could replace with other position-wise operations

In the Bigger Picture

```
Transformer Layer:
├─ Attention:       Cross-position mixing    (global, content-based)
└─ FFN (1×1 conv):  Position-wise processing (local,  independent)
```

Thus Transformers are both powerful and interpretable!

Transformer Decoder Layer

More complex than encoder:

1. **Masked Self-Attention:**
   - Prevents attending to future tokens
   - Preserves autoregressive property
2. **Cross-Attention:**
   - Queries from decoder
   - Keys and Values from encoder output
   - Connects source and target
3. **Feed-Forward:**
   - Same as encoder

Information flow in decoder

Decoder Input
↓
┌──────────────────────────────────┐
│ 1. Masked Self-Attention │ ← Look at previous decoder tokens
│ Q, K, V from decoder │
│ + Add & Norm │
└──────────────────────────────────┘
↓
┌──────────────────────────────────┐
│ 2. Cross-Attention │ ← ENCODER OUTPUT enters here
│ Q: decoder │
│ K, V: ENCODER ←──│─── Encoder Output (Keys & Values)
│ + Add & Norm │
└──────────────────────────────────┘
↓
┌──────────────────────────────────┐
│ 3. Feed-Forward Network │ ← Position-wise processing

│ + Add & Norm │
└──────────────────────────────────────┘

↓
Layer Output

How Encoder's Output is Used in Decoder

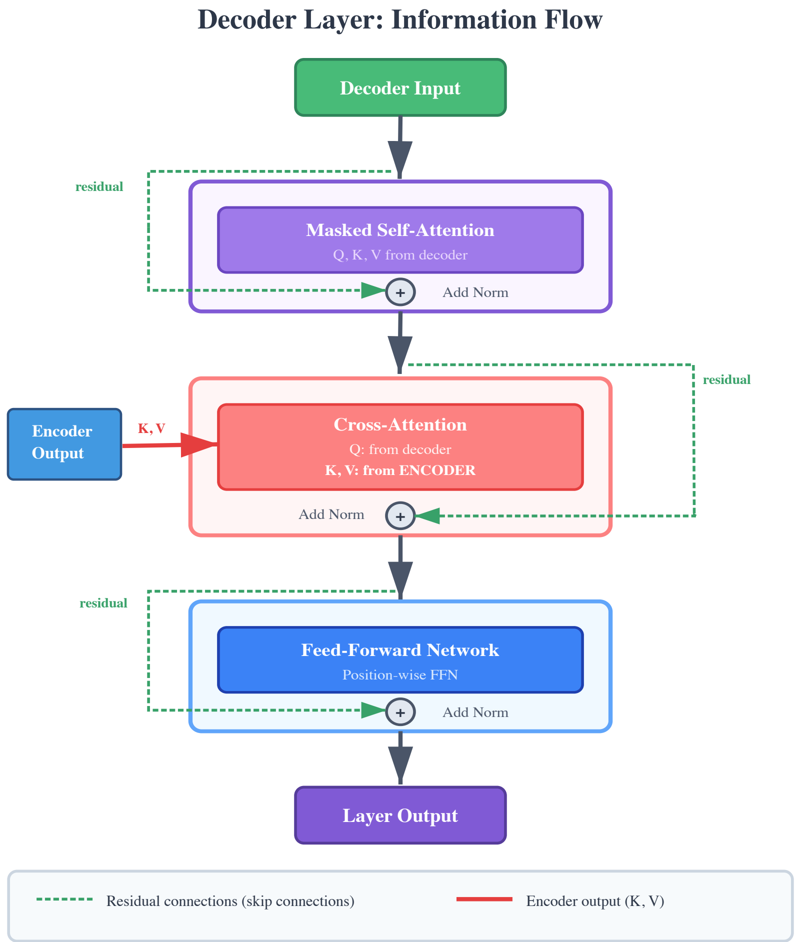
The encoder output is used in the **cross-attention** (encoder-decoder attention) sublayer of each decoder layer

- **Encoder output provides Keys (K) and Values (V)**
- **Decoder provides Queries (Q)**

Detailed Architecture Flow

```
ENCODER (processes source: "The cat sat")
   ↓
encoder_output = [h₁, h₂, h₃]  (one vector per source token)
   ↓
   ↓ (This gets fed to EVERY decoder layer)
   ↓
   ↓
DECODER (generates target: "Le chat")

   Decoder Layer (repeated N times):
   ┌──────────────────────────────────────┐
   │ 1. Masked Self-Attention             │
   │    (decoder tokens attend to         │
   │     previous decoder tokens)         │
   │    Q, K, V all from decoder          │
   └──────────────────────────────────────┘
              ↓ (residual + norm)
   ┌──────────────────────────────────────┐
   │ 2. Cross-Attention  ← ENCODER HERE!  │
   │    Q: from decoder current state     │
   │    K: from encoder_output            │
   │    V: from encoder_output            │
   │                                      │
   │    Attention(Q_dec, K_enc, V_enc)    │
   └──────────────────────────────────────┘
              ↓ (residual + norm)
   ┌──────────────────────────────────────┐
   │ 3. Feed-Forward Network              │
   └──────────────────────────────────────┘
              ↓ (residual + norm)
```



The Math: How It's Combined

```
# In each decoder layer:

# Step 1: Masked self-attention
x = decoder_input  # Current decoder state
self_attn_out = MaskedSelfAttention(Q=x, K=x, V=x)
x = LayerNorm(x + self_attn_out)  # Residual connection

# Step 2: Cross-attention (ENCODER OUTPUT USED HERE!)
cross_attn_out = CrossAttention(
    Q=x,                    # Query from current decoder state
    K=encoder_output,       # Keys from encoder ← HERE
    V=encoder_output        # Values from encoder ← HERE
)
x = LayerNorm(x + cross_attn_out)  # Residual connection (ADDED)

# Step 3: Feed-forward
ff_out = FeedForward(x)
x = LayerNorm(x + ff_out)  # Residual connection
```

Key

1. Encoder's output is ADDED via residual connection

- just like self-attention output

2. The encoder_output is used as K and V

```
# Cross-attention projects encoder_output
K = encoder_output @ W_K  # Keys from source
V = encoder_output @ W_V  # Values from source
```

```python
Q = decoder_state @ W_Q   # Query from target

# Then standard attention
attention_weights = softmax(Q @ K.T / sqrt(d_k))
output = attention_weights @ V
```

3. Same encoder_output used in ALL decoder layers

- Each decoder layer has its own cross-attention
- But they all attend to the same encoder_output
  - only the final encoder output is added to each decoder layer
  - there is a key, value pair for each token
  - only some architectures add key-value pair on a encoder-decoder layer-per-layer basis
    - these are not standard Transformers
    - a very high number of parameters, like a dense-net
  - simple and flexible architecture
  - good information flow
  - learning is not incremental
  - layers might focus on different semantic relationships and long-range dependencies
- Different layers learn different attention patterns

4. The encoder_output is a SEQUENCE

```python
encoder_output.shape = (batch_size, source_seq_len, d_model)
# One vector for each source token
# Decoder can attend to any/all source positions
```

Complete PyTorch Example

```python
class TransformerDecoderLayer(nn.Module):
    def __init__(self, d_model=512, num_heads=8, d_ff=2048, dropout=0.1):
        super().__init__()

        # 1. Masked self-attention
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.norm1 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)

        # 2. Cross-attention (encoder-decoder attention)
        self.cross_attn = MultiHeadAttention(d_model, num_heads)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout2 = nn.Dropout(dropout)

        # 3. Feed-forward
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model)
        )
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout3 = nn.Dropout(dropout)

    def forward(self, x, encoder_output, src_mask=None, tgt_mask=None):
        """
        Args:
            x: Decoder input (batch, target_seq_len, d_model)
            encoder_output: Output from encoder (batch, source_seq_len, d_model)
            src_mask: Mask for source sequence (padding)
            tgt_mask: Causal mask for target sequence
        """

        # 1. Masked self-attention (decoder attends to itself)
        self_attn_out = self.self_attn(
```

```
            Q=x, K=x, V=x,
            mask=tgt_mask
        )
        x = self.norm1(x + self.dropout1(self_attn_out))  # Residual + norm

        # 2. Cross-attention (decoder attends to encoder)
        cross_attn_out = self.cross_attn(
            Q=x,                    # Query from decoder
            K=encoder_output,        # Keys from encoder
            V=encoder_output,        # Values from encoder
            mask=src_mask
        )
        x = self.norm2(x + self.dropout2(cross_attn_out))  # Residual + norm (ADDED)

        # 3. Feed-forward
        ff_out = self.ffn(x)
        x = self.norm3(x + self.dropout3(ff_out))  # Residual + norm

        return x
```

Why This Design?

Query from decoder = "What do I need to generate this target word?"

- Based on what decoder has generated so far
- Changes at each decoding step

Keys & Values from encoder = "Here's what the source sentence contains"

- Fixed encoding of source
- Decoder queries it to find relevant information

Example: Translating "The cat sat" → "Le chat"

When generating "chat":

```
# Decoder state (after generating "Le"):
Q_decoder = "I need to translate the main subject noun"

# Encoder provides:
K_encoder = ["determiner", "ANIMAL", "action", ...]  # Keys for matching
V_encoder = [semantic_the, semantic_cat, semantic_sat, ...]  # Content

# Cross-attention computes:
attention = softmax(Q_decoder @ K_encoder.T)  # High weight on "cat"
output = attention @ V_encoder  # Retrieve cat semantics

# This output is ADDED to decoder state via residual
decoder_state = decoder_state + output  # Enriched with source info
```

- Encoder output is used in the **cross-attention sublayer** of each decoder layer
- It provides the **Keys and Values** (decoder provides Query)
- The cross-attention output is **ADDED** to the decoder state via residual connection
- Same encoder output is used by all decoder layers (each learns different attention patterns)

This design **allows the decoder to dynamically "look at" different parts of the source** sequence at each generation step, deciding what source information is relevant for generating each target token.

Masking in Transformer

Padding Mask:

```
# Don't attend to padding tokens
mask = (input_tokens == PAD_TOKEN)
scores = scores.masked_fill(mask, -1e9)
```

Causal Mask (Look-ahead mask):

```
# Prevent attending to future tokens
mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1).bool()
#    t0  t1  t2  t3
# t0 [0,  1,  1,  1]  # Can only see t0
# t1 [0,  0,  1,  1]  # Can see t0, t1
# t2 [0,  0,  0,  1]  # Can see t0, t1, t2
# t3 [0,  0,  0,  0]  # Can see all

scores = scores.masked_fill(mask, -1e9)
attention = softmax(scores)  # Masked positions → 0 after softmax
```

---

7. Positional Encoding

The Position Problem

Attention is permutation-equivariant:

$$\text{Attention}(\pi(X)) = \pi(\text{Attention}(X))$$

for any permutation π

This means:

```
# Without positional info, these are identical:
["cat", "sat", "mat"] ≡ ["mat", "cat", "sat"]
```

But word order matters!

- "cat sat on mat" ≠ "mat sat on cat"

Sinusoidal Positional Encoding

Formula:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

where:

- pos: position in sequence (0, 1, 2, ...)
- i: dimension index (0, 1, ..., $d_{model}/2$)
- Even dimensions use sine, odd use cosine

Key Properties:

1. **Unique encoding for each position**
2. **Relative position information**: PE_{pos+k} is a linear function of PE_{pos}
3. **Bounded values**: $[-1, 1]$
4. **No learned parameters**: Deterministic function
5. in original Transformer

Adding or concatenating?

1. the new vector may be concatenated
    1. this extends the dimensionality of encoding
    2. if one-hot encoding of dimensions, the dimension grows quickly
    3. but is reversible
2. or it may be added
    1. each position is differently coded
    2. irreversible
    3. modified when learning

Why Sinusoidal?

Mathematical Insight:

For relative position k:

$$PE(pos + k) = T_k \cdot PE(pos)$$

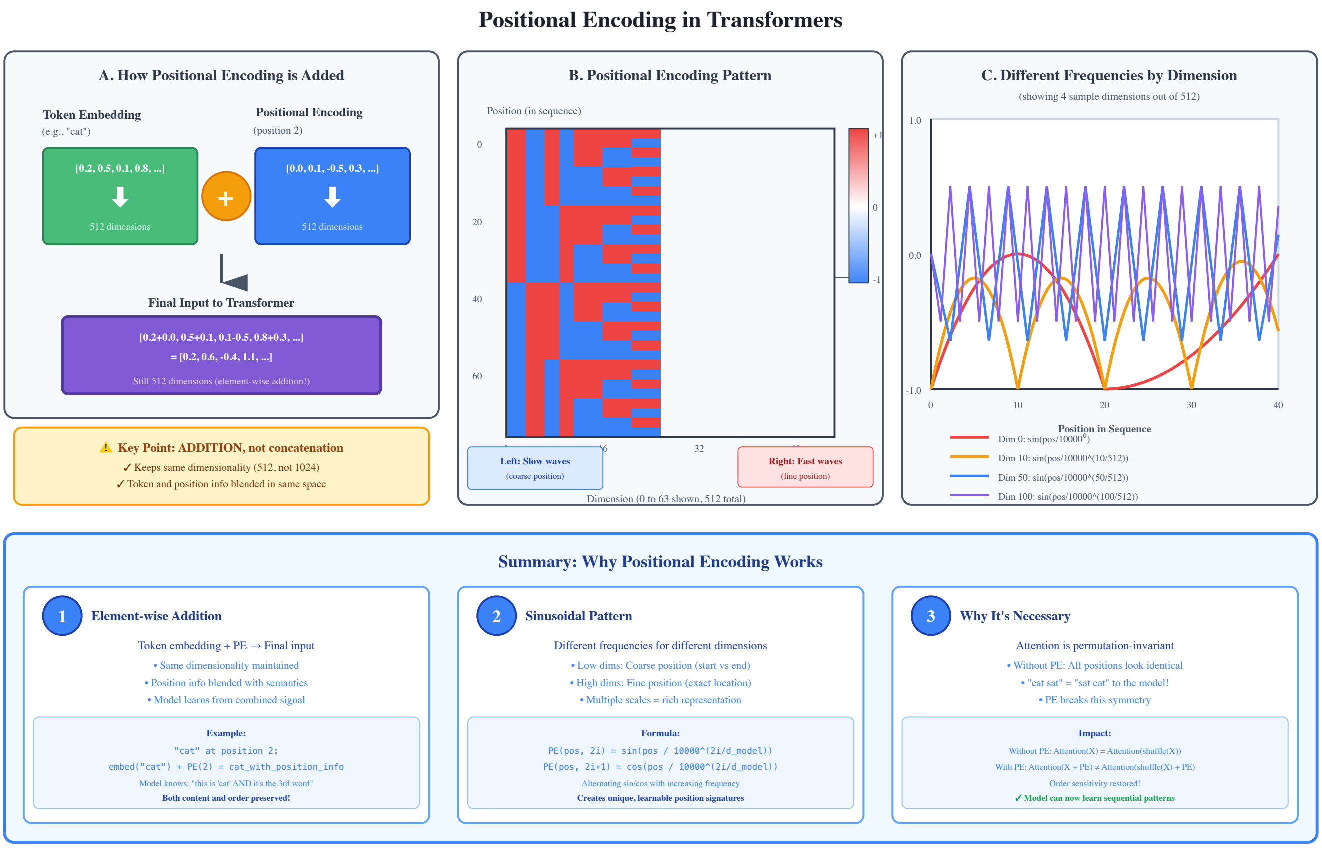
where T_k is a transformation matrix that depends only on k

$$\sin(\alpha + \beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

This allows model to learn to attend by relative position!

Alternative: Learned Positional Embeddings

```
self.pos_embedding = nn.Embedding(max_seq_len, d_model)
# Trade-off: More flexible but can't extrapolate to longer sequences
```



2D Positional Encoding

```python
def create_2d_position_encoding(h, w, embed_dim):
    """Separate encodings for x and y coordinates"""
    pos_embed_h = sinusoidal_position_encoding(h, embed_dim // 2)
    pos_embed_w = sinusoidal_position_encoding(w, embed_dim // 2)

    # Combine row and column encodings
    pos_embed = torch.cat([
        pos_embed_h.unsqueeze(1).repeat(1, w, 1),
        pos_embed_w.unsqueeze(0).repeat(h, 1, 1)
    ], dim=-1)

    return pos_embed.flatten(0, 1)  # (h*w, embed_dim)
```

Relative Position Encoding

Instead of absolute positions, encode relative distances:

$$\text{Attention}_{ij} = \text{softmax}\left(\frac{q_i \cdot k_j + r_{ij}}{\sqrt{d}}\right)$$

where r_{ij} encodes the relative position between patches i and j

---

Learnable Position Embeddings (ViT Default)

```python
self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + 1, embed_dim))
nn.init.trunc_normal_(self.pos_embed, std=0.02)

# Add to patch embeddings
x = x + self.pos_embed
```

Advantages:

- Flexible, can learn any pattern
- Simple to implement
- Empirically effective

Disadvantages:

- Fixed resolution at training
- Doesn't naturally generalize to different image sizes

---

8. Why Transformers Work: Theoretical Insights

Path Length Between Dependencies

Architecture	Max Path Length	Complexity per Layer	Sequential Ops
RNN	$O(n)$	$O(n \cdot d^2)$	$O(n)$
CNN	$O(\log_k n)$	$O(k \cdot n \cdot d^2)$	$O(1)$
Transformer	$O(1)$	$O(n^2 \cdot d)$	$O(1)$

Key Advantage:

- Any two positions connected in $O(1)$ layers
- Direct gradient paths for all pairs
- Enables learning long-range dependencies

The Transformer as a Composition of Functions

Mathematical View:

A Transformer is:

$$f(x) = f_N \circ f_{N-1} \circ \cdots \circ f_2 \circ f_1(x)$$

where each layer f_i is:

$$f_i(x) = x + \text{FFN}(x + \text{Attention}(x))$$

remember:

- **Attention**: Routes information (soft routing)
- **FFN**: Processes information (computation)
- **Residual**: Preserves gradient flow

Attention as Soft Dictionary Lookup (Formal)

Theorem: Attention can implement any dictionary lookup

Given keys $K = [k_1, \ldots, k_m]$ and values $V = [v_1, \ldots, v_m]$:

Hard lookup:

$$\text{lookup}(q) = v_i \quad \text{where} \quad i = \arg\max_j \langle q, k_j \rangle$$

Soft lookup (Attention):

$$\text{Attention}(q) = \sum_j \text{softmax}(\langle q, k_j \rangle / \tau) \cdot v_j$$

As $\tau \to 0$ (temperature), $\text{Attention}(q) \to \text{lookup}(q)$

Generalization:

- Multiple queries in parallel: $Q = [q_1, \ldots, q_n]$
- Batch processing all lookups simultaneously

Transformer as Universal Approximator

Theorem (Yun et al., 2020):

A Transformer with sufficient depth and width can approximate any sequence-to-sequence function to arbitrary precision.

Key components:

1. **Multi-head attention**: Can implement any sparse connectivity pattern
2. **FFN with ReLU**: Universal function approximation
3. **Depth**: Compositional representations
4. **Residual connections**: Information highways

Intuition:

```
# Can approximate any f: sequence → sequence
y = Transformer(x)

# By composing:
y = FFN_N(Attn_N(...FFN_1(Attn_1(x))))
```

Why Residual Connections Matter

Standard Network:

$$x_{l+1} = f_l(x_l)$$

With Residuals:

$$x_{l+1} = x_l + f_l(x_l)$$

Gradient Flow:

$$\frac{\partial \mathcal{L}}{\partial x_l} = \frac{\partial \mathcal{L}}{\partial x_{l+1}} \left(1 + \frac{\partial f_l}{\partial x_l}\right)$$

The "$+1$" ensures:

- Gradients can't vanish (always have direct path)
- Network can learn identity if needed: $f_l(x) = 0$
- Enables very deep networks (100+ layers)

Layer Normalisation: Why It's Critical

Formula:

$$\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sigma + \epsilon} + \beta$$

where

$$\begin{aligned} \mu &= \frac{1}{d} \sum_i x_i \\ \sigma^2 &= \frac{1}{d} \sum_i (x_i - \mu)^2 \end{aligned}$$

Why in Transformers?

1. **Stabilises training**: Prevents activation explosion
2. **Enables deeper networks**: Each layer starts with normalised distribution
3. **Reduces dependence on initialisation**: Less sensitive to weight initialisation
4. **Faster convergence**: Smoother loss landscape

Pre-LN vs Post-LN:

```
# Post-LN (original Transformer)
x = LayerNorm(x + Sublayer(x))

# Pre-LN (modern, more stable)
x = x + Sublayer(LayerNorm(x))
```

Modern Transformers predominantly use Pre-LN!

---

9. Representing Data as Sequences

The Universal Sequence View

Core Idea: Almost any structured data can be represented as a sequence

```
Text:    "Hello world"          → [h, e, l, l, o, _, w, o, r, l, d]
Images:  224×224 image          → 196 patches of 16×16
Graphs:  Social network         → [node1, node2, ..., nodeN]
```

```
Audio:   Waveform                → [sample1, sample2, ...]
Video:   Frame sequence          → [frame1, frame2, ...]
```

Key insight: Sequence processing is a general framework!

Tokenisation: The First Step

What is a Token?

- Atomic unit of input that Transformer processes
- Can be words, subwords, characters, patches, nodes, etc.

Common Tokenization Strategies:

Domain	Tokens	Example
NLP	Words	["The", "cat", "sat"]
NLP	Subwords (BPE)	["The", "cat", "s", "at"]
Vision	Patches	16×16 pixel patches
Audio	Spectrograms	Time-frequency bins
Graphs	Nodes	Graph vertices
Code	Tokens	["def", "function", "(", "x", ")"]

Text as Sequences: NLP

Example: Machine Translation

```
# Tokenization
source = "The cat sat on the mat"
tokens = tokenizer(source)  # ["The", "cat", "sat", "on", "the", "mat"]

# Embedding
embeddings = embedding_layer(tokens)  # (6, 512)

# Add positional encoding
embeddings = embeddings + positional_encoding

# Process with Transformer
output = transformer(embeddings)

# Decode
translation = decoder(output)  # "Le chat s'est assis sur le tapis"
```

Why it works:

- Natural sequential structure
- Order matters (syntax, semantics)
- Long-range dependencies (anaphora, discourse)

Images as Sequences: Vision

Patching Strategy (covered in ViT lecture):

```
# Image: 224×224×3
# Divide into 16×16 patches
# Result: 14×14 = 196 patches

patches = rearrange(image, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)',
```

```
                     p1=16, p2=16)
# patches: (batch, 196, 768)
```

Sequential View:

```
Position:   [1]    [2]    [3]   ...  [196]
             ↓      ↓      ↓           ↓
Patches:   [P₁]   [P₂]   [P₃]   ... [P₁₉₆]
```

Each patch is a "word" in the image "sentence"!

Graphs as Sequences: Graph Neural Networks

Challenge: Graphs don't have natural order

Solutions:

1. **Arbitrary Ordering:**

```
# Order nodes arbitrarily
nodes = [n1, n2, n3, ..., nN]
# Use attention to learn relationships
# Positional encoding less meaningful
```

2. **Adjacency-Based Masking:**

```
# Only attend to neighbors
attention_mask[i, j] = 1 if edge(i, j) exists
                       0 otherwise
```

3. **Graph Transformer:**

```
# Structural encoding
structure_encoding = encode_graph_structure(adj_matrix)
node_features = node_features + structure_encoding
output = transformer(node_features)
```

Time Series and Audio

Raw Audio:

```
# Waveform: sequence of samples
audio = [sample_1, sample_2, ..., sample_T]
# Direct application: T can be very large!
```

Spectrogram:

```
# 2D representation: frequency × time
spectrogram = STFT(audio)  # (freq_bins, time_frames)
# Treat as sequence of frequency vectors
tokens = [freq_vec_1, freq_vec_2, ..., freq_vec_T]
```

Hierarchical Processing:

```
# Multi-scale: subsample at different rates
coarse = audio[::100]  # Every 100th sample
medium = audio[::10]   # Every 10th sample
fine = audio          # All samples
```

Multimodal: Combining Different Modalities

Vision + Language (e.g., CLIP, DALL-E):

```
# Concatenate sequences from different modalities
image_tokens = vision_encoder(image)     # (196, 512)
text_tokens = text_encoder(text)         # ( 20, 512)

# Unified sequence
combined = concatenate([image_tokens, text_tokens])  # (216, 512)

# Process with Transformer
output = transformer(combined)

# Cross-modal attention!
# Image tokens can attend to text tokens and vice versa
```

Key Insight:

- Different modalities → Different tokenizations
- But same attention mechanism!
- Shared representation space

Why Sequence Representation is Powerful

1. **Unified Framework**: Same architecture for different data types
2. **Flexible Interactions**:
    - Self-attention within modality
    - Cross-attention between modalities
3. **Compositionality**:
    - Complex structures from simple tokens
    - Hierarchical relationships emerge
4. **Scalability**:
    - Parallel processing of all tokens
    - GPU-friendly computation

Trade-off:

- Lose domain-specific inductive biases
- Need more data to learn structure
- But gain flexibility and generality!

---

10. Architecture Details

Complete Transformer Encoder

$$\begin{aligned} Z &= \text{LayerNorm}(X + \text{MultiHeadAttention}(X, X, X)) \\ \text{Output} &= \text{LayerNorm}(Z + \text{FFN}(Z)) \end{aligned}$$

Component Breakdown:

```
class TransformerEncoder(nn.Module):
    def __init__(self, d_model=512, num_heads=8, d_ff=2048, dropout=0.1):
        super().__init__()

        # Multi-head attention
        self.self_attn = MultiHeadAttention(d_model, num_heads)

        # Feed-forward network
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff),
```

```
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model)
        )

        # Layer normalization
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

        # Dropout
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Self-attention with residual
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout1(attn_output))

        # Feed-forward with residual
        ffn_output = self.ffn(x)
        x = self.norm2(x + self.dropout2(ffn_output))

        return x
```

Complete Transformer Decoder

```
class TransformerDecoder(nn.Module):
    def __init__(self, d_model=512, num_heads=8, d_ff=2048, dropout=0.1):
        super().__init__()

        # Masked self-attention (causal)
        self.self_attn = MultiHeadAttention(d_model, num_heads)

        # Cross-attention to encoder
        self.cross_attn = MultiHeadAttention(d_model, num_heads)

        # Feed-forward
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model)
        )

        # Layer norms
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)

        # Dropout
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)
        self.dropout3 = nn.Dropout(dropout)

    def forward(self, x, encoder_output,
                src_mask=None, tgt_mask=None):
        # Masked self-attention
        attn1 = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout1(attn1))

        # Cross-attention to encoder
        attn2 = self.cross_attn(x, encoder_output,
                                encoder_output, src_mask)
        x = self.norm2(x + self.dropout2(attn2))

        # Feed-forward
        ffn_out = self.ffn(x)
        x = self.norm3(x + self.dropout3(ffn_out))
```

```python
        return x
```

Training Details: Teacher Forcing

During Training:

```python
# Use ground truth as input (parallel training)
# Input:  [SOS, "Le", "chat", "s'est"]
# Target: ["Le", "chat", "s'est", "assis", "EOS"]

for batch in dataloader:
    encoder_out = encoder(source)

    # Decoder sees ground truth (shifted right)
    decoder_out = decoder(target[:-1], encoder_out)

    # Predict next token
    loss = criterion(decoder_out, target[1:])

    loss.backward()
    optimizer.step()
```

During Inference:

```python
# Autoregressive generation (sequential)
generated = [SOS]

for _ in range(max_length):
    decoder_out = decoder(generated, encoder_out)
    next_token = argmax(decoder_out[-1])

    if next_token == EOS:
        break

    generated.append(next_token)
```

Practical Hyperparameters

Original Transformer ("Attention is All You Need"):

```python
config = {
    'd_model': 512,           # Model dimension
    'num_layers': 6,          # Encoder and decoder layers
    'num_heads': 8,           # Attention heads
    'd_ff': 2048,             # FFN inner dimension
    'dropout': 0.1,           # Dropout rate
    'max_seq_len': 512,       # Maximum sequence length
    'vocab_size': 37000,      # Vocabulary size
}
```

Transformer-Big (Better performance):

```python
config_big = {
    'd_model': 1024,
    'num_layers': 6,
    'num_heads': 16,
    'd_ff': 4096,
    'dropout': 0.3,
}
```

Modern Large Models (GPT-3, etc.):

```
config_large = {
    'd_model': 12288,          # 12K dimensions!
    'num_layers': 96,          # 96 layers
    'num_heads': 96,
    'd_ff': 49152,
    'params': '175B',          # 175 billion parameters
}
```

Optimization: Learning Rate Schedule

Warmup + Decay:

```
def get_lr(step, d_model=512, warmup_steps=4000):
    arg1 = step ** (-0.5)
    arg2 = step * (warmup_steps ** (-1.5))
    return (d_model ** (-0.5)) * min(arg1, arg2)
```

$$\text{lr} = d_{model}^{-\frac{1}{2}} \cdot \min(\text{step}^{-\frac{1}{2}}, \text{step} \cdot (\text{warmup-steps}^{-\frac{3}{2}}))$$

$$\text{lr} = \frac{1}{\sqrt{d_{model}}} \cdot \min\left(\frac{1}{\sqrt{\text{step}}}, \text{step} \cdot \left(\frac{1}{\sqrt{\text{warmup-steps}^3}}\right)\right)$$

Why Warmup?

- Transformers sensitive to initialization
- Large gradients early in training
- Warmup stabilizes training

---

11. Computational Complexity Analysis

Attention Complexity

Standard Self-Attention:

$$\text{Complexity} = O(n^2 \cdot d)$$

where:

- n: sequence length
- d: model dimension

Breakdown:

1. QK^T: $O(n^2 \cdot d)$ — matrix multiplication
2. Softmax: $O(n^2)$ — row-wise operation
3. Multiply by V: $O(n^2 \cdot d)$

Total: $O(n^2 \cdot d)$ per layer

For full Transformer:

- Encoder: $L \cdot O(n^2 \cdot d)$ where L = number of layers
- Decoder: Similar but with additional cross-attention

Memory Complexity

Storing Attention Matrices:

$$\text{Memory} = O(n^2 \cdot h + n \cdot d)$$

where h is number of heads

Components:

- Attention weights: $h \times n \times n$ (can be large!)
- Activations: $n \times d$ per layer
- Gradients: Same as activations (during training)

Example:

```
# Sequence length n = 1024, d_model = 512, heads = 8
attention_matrix = 8 * 1024 * 1024 * 4 bytes  # ~33 MB per layer
# For 12 layers: ~400 MB just for attention!
```

Comparison with Other Architectures

Architecture	Complexity per Layer	Sequential Ops	Max Path Length
RNN	$O(n \cdot d^2)$	$O(n)$	$O(n)$
CNN	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$

Trade-offs:

- RNN: Sequential bottleneck, but linear memory
- CNN: Limited receptive field, but efficient
- Transformer: $O(n^2)$ complexity, but parallel + global

Efficient Attention Variants

Problem: $O(n^2)$ doesn't scale to very long sequences

Solutions:

1. **Sparse Attention** (Routing patterns):

```
# Only attend to local neighbors + few global tokens
attention_pattern = local_window + strided + global_tokens
# Reduces to O(n * k) where k << n
```

2. **Linear Attention** (Kernelized):

```
# Approximate attention with kernel functions
# O(n) complexity!
attention = φ(Q) @ (φ(K)^T @ V)
# instead of (Q @ K^T) @ V
```

3. **Flash Attention** (I/O efficient):

```
# Fused CUDA kernels
# Same complexity but much faster in practice
# Reduces memory from O(n²) to O(n)
```

---

12. Why Transformers Dominate

Advantages Over Previous Architectures

Aspect	RNN/LSTM	CNN	Transformer
Parallelization	✗ Sequential	✓ Parallel	✓ Parallel
Long-range deps	✗ Gradient vanishing	✗ Limited receptive field	✓ Direct connections
Inductive bias	Strong (sequential)	Strong (locality)	Weak (flexible)
Data efficiency	✓ Good	✓ Good	✗ Needs large data
Interpretability	✗ Hidden state	✗ Feature maps	✓ Attention weights
Scalability	✗ Limited	✗ Saturates	✓ Continues improving

The Scaling Hypothesis

Key Observation: Transformers continue to improve with:

- More parameters
- More data
- More compute

```
Performance = Power Law(Scale)

log(Loss) ≈ −α ∗ log(N) + C

where N = parameters, data, or compute
```

This doesn't hold for CNNs/RNNs! They saturate.

Empirical Wins

NLP:

- Machine Translation: BLEU score improvements
- Language Modeling: Perplexity reductions
- Few-shot Learning: Emergent capabilities

Vision:

- ImageNet: State-of-the-art accuracy
- Object Detection: Superior to CNN-based methods
- Video Understanding: Temporal modeling

Multi-modal:

- CLIP: Zero-shot classification
- DALL-E: Text-to-image generation
- Flamingo: Few-shot learning across modalities

The Lottery: What Makes Transformers Special?

Hypotheses:

1. **Inductive Bias Trade-off**:
   - Weak priors allow learning from data
   - Doesn't impose wrong assumptions
2. **Expressiveness**:
   - Universal approximation with fewer constraints
   - Can represent more complex functions
3. **Optimisation Landscape**:
   - Residual connections create smooth paths

- Skip connections prevent gradient issues

4. **Attention as Routing**:
   - Soft, learnable connectivity
   - Adaptive computation based on content
5. **Parallel Training**:
   - Efficient use of modern hardware
   - Scales better with resources

---

13. Limitations and Challenges

Computational Cost

```
# Example: GPT-3 training
parameters = 175e9
tokens = 300e9
compute = 3.14e23 FLOPs  # ~$4.6M in cloud costs!
training_time = ~1 month on thousands of GPUs
```

Inference Cost:

```
# Single forward pass for GPT-3
sequence_length = 2048
floating_point_ops = 2 * 175e9 * 2048  # ~700 GFLOPs
# Hundreds of ms latency
```

Quadratic Complexity in Sequence Length

Problem:

```
# Memory and compute grow as O(n²)
seq_len = [128, 256, 512, 1024, 2048, 4096]
memory = [x**2 for x in seq_len]
# [16K, 65K, 262K, 1M, 4M, 16M] elements

# Can't process very long sequences!
# Books, long documents, high-res images, videos
```

Partial Solutions:

- Sparse attention patterns
- Linear attention approximations
- Hierarchical processing
- Segmentation strategies

Data Hunger

Observation: Transformers need massive amounts of data

```
# Typical requirements
small_model = {
    'params': '110M',
    'data': '10B tokens',  # ~10GB text
    'compute': '1e20 FLOPs'
}

large_model = {
    'params': '175B',
    'data': '300B tokens',  # ~300GB text
```

```
    'compute': '3e23 FLOPs'
}
```

Why?

- Weak inductive biases
- Must learn structure from data
- Overparameterization requires regularization through data

Interpretability Challenges

Attention ≠ Explanation:

```
# Common misconception:
# "High attention weight means the model uses this information"

# Reality:
# - Attention is just one component
# - FFN can override attention
# - Multiple heads complicate interpretation
# - Attention can be uniform but still useful
```

Open Questions:

- What do different heads learn?
- Why do some heads become "no-op"?
- How does information flow through layers?
- What concepts do neurons represent?
- how is knowledge actually stored?

---

14. Modern Variants and Extensions

Encoder-Only Models

BERT (Bidirectional Encoder Representations from Transformers):

```
# Masked language modeling
input:  ["The", "[MASK]", "sat", "on", "the", "mat"]
output: ["The", "cat", "sat", "on", "the", "mat"]

# Used for:
# - Classification
# - Named Entity Recognition
# - Question Answering
```

Architecture:

- Only encoder stack
- Bidirectional attention (see full context)
- Pre-trained on massive corpora

Decoder-Only Models

GPT (Generative Pre-trained Transformer):

```
# Autoregressive language modeling
input:  ["The", "cat", "sat"]
output: "on"  # Predict next token

# Used for:
```

```
# - Text generation
# - Few-shot learning
# - In-context learning
```

Architecture:

- Only decoder stack (with causal masking)
- Unidirectional attention
- Scales to massive sizes (GPT-3: 175B params)

Key Insight: Decoder-only models can do everything!

- Generation (natural)
- Classification (with prompting)
- Translation (with examples)

Efficient Transformers

Reformer (Kitaev et al., 2020):

- Locality-sensitive hashing for attention
- Reversible layers (saves memory)
- $O(n \log n)$ complexity

Linformer (Wang et al., 2020):

- Low-rank approximation of attention
- $O(n)$ complexity
- Small accuracy drop

Performer (Choromanski et al., 2021):

- Kernelized attention (FAVOR+)
- $O(n)$ complexity
- Maintains performance

---

15. Summary & Key Takeaways

Core Concepts Recap

1. **Attention Mechanism**:

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

   - Soft, differentiable routing of information
   - Content-based, not position-based
2. **Transformer Architecture**:
   - Stacked encoder-decoder with attention
   - Multi-head attention for diverse patterns
   - Position-wise feed-forward networks
   - Residual connections + Layer normalization
3. **Why It Works**:
   - $O(1)$ path between any two positions
   - Parallel processing (efficient training)
   - Flexible, learned inductive biases
   - Scales with compute and data

When to Use Transformers

✅ **Use Transformers When:**

- Large dataset available
- Long-range dependencies important
- Parallel training resources available
- Flexibility more important than efficiency
- State-of-the-art performance needed

❌ **Consider Alternatives When:**

- Small dataset (<10K examples)
- Real-time inference critical
- Memory/compute constrained
- Strong domain priors available
- Interpretability is paramount

The Road Ahead

Current Research Directions:

1. **Efficiency**:
   - Linear attention mechanisms
   - Sparse transformers
   - Model compression
2. **Long Context**:
   - Extending to 100K+ tokens
   - Hierarchical processing
   - Memory-augmented transformers
3. **Multimodal**:
   - Unified architectures
   - Cross-modal learning
   - Few-shot transfer
4. **Theory**:
   - Understanding what is learned
   - Optimization dynamics
   - Generalization bounds

---

16. Practical Tips for Implementation

Debugging Checklist

```
# Common issues and solutions

1. **Exploding/Vanishing Gradients**
   - Check: Gradient norms
   - Fix: Gradient clipping, learning rate warmup

2. **Attention Collapse**
   - Check: Attention weight entropy
   - Fix: Dropout, smaller initialization

3. **Training Instability**
   - Check: Loss spikes
   - Fix: Mixed precision, gradient accumulation

4. **Poor Performance**
   - Check: Positional encoding, masking
   - Fix: Verify masks, try learned positions
```

```
5. **OOM (Out of Memory)**
   - Check: Batch size, sequence length
   - Fix: Gradient checkpointing, smaller batches
```

Hyperparameter Tuning Guide

Start Here (Defaults):

```
config = {
    'learning_rate': 1e-4,
    'warmup_steps': 4000,
    'batch_size': 32,
    'dropout': 0.1,
    'num_layers': 6,
    'num_heads': 8,
    'd_model': 512,
    'd_ff': 2048,
}
```

Tuning Priority:

1. Learning rate (most impactful)
2. Warmup steps
3. Batch size
4. Dropout
5. Architecture (last resort)

Pre-training vs Fine-tuning

Pre-training (if you have massive compute):

```
# Train from scratch on large corpus
# Requires: 100GB+ data, weeks of training, $$$

model = Transformer(...)
train(model, large_corpus, epochs=100)
```

Fine-tuning (recommended for most):

```
# Start from pre-trained model
# Requires: Small dataset, hours of training, $

model = load_pretrained('bert-base')
model.classifier = nn.Linear(768, num_classes)

# Lower learning rate for fine-tuning
optimizer = AdamW(model.parameters(), lr=2e-5)
train(model, task_data, epochs=3)
```

---

17. Connections to Future Topics

What's Next in This Course

Week 6: Sequence Models (RNNs, LSTMs, S4)

- How Transformers evolved from RNNs
- Trade-offs between architectures
- When to use what

Week 7: Self-Supervised Learning

- BERT and GPT pre-training objectives
- Contrastive learning with Transformers
- Masked modeling strategies

Week 10: Scaling Laws

- Why Transformers scale so well
- Emergent abilities
- Data/compute trade-offs

Week 12: Mechanistic Interpretability

- What Transformers learn
- Attention patterns
- Circuit discovery

---

18. References & Further Reading

Essential Papers

1. **Attention Is All You Need** (Vaswani et al., 2017)
   - The original Transformer paper
   - Must-read foundation
2. **BERT** (Devlin et al., 2018)
   - Bidirectional pre-training
   - Encoder-only architecture
3. **GPT-2** and **GPT-3** (OpenAI)
   - Decoder-only at scale
   - Few-shot learning emergence
4. **Vision Transformer (ViT)** (Dosovitskiy et al., 2020)
   - Covered in Lecture 04
   - Transformers for images
5. **Formal Algorithms for Transformers** (Phuong & Hutter, 2022)
   - Comprehensive mathematical treatment
   - Excellent reference

Tutorials and Code

- **The Annotated Transformer**
  - Line-by-line implementation
  - Best starting point for coding
- **Hugging Face Transformers**
  - Production-ready implementations
  - Pre-trained models
- **PyTorch Transformer Tutorial**
  - Official tutorial
  - Clean implementation
- **minGPT**
  - Minimal GPT implementation
  - By Andrej Karpathy

Advanced Topics

- **Efficient Transformers**: Survey Paper

- **Scaling Laws**: Kaplan et al.
- **Interpretability**: A Mathematical Framework for Transformer Circuits
- **Optimization**: On Layer Normalization

Appendix: Mathematical Derivations

A1: Why Scale by √d_k?

Problem: Without scaling, dot products grow large in high dimensions

Proof: Assume $q_i, k_j \ \mathcal{N}(0, 1)$ independently.

$$q \cdot k = \sum_{i=1}^{d_k} q_i k_i$$

$$[q \cdot k] = \sum_{i=1}^{d_k} [q_i][k_i] = 0$$

$$\mathrm{ar}(q \cdot k) = \sum_{i=1}^{d_k} \mathrm{ar}(q_i k_i) = d_k$$

So $q \cdot k \ \mathcal{N}(0, d_k)$

Issue: Softmax saturates when inputs are large!

$$\text{f } x \ \ 0: \quad \mathrm{softmax}(x) \ \ [0, \ldots, 1, \ldots, 0]$$

(near one-hot → small gradients)

Solution: Scale by $\sqrt{d_k}$:

$$\frac{q \cdot k}{\sqrt{d_k}} \ \ \mathcal{N}(0, 1)$$

Now variance is constant regardless of dimension!

A2: Positional Encoding Properties

Claim: For offset k, PE_{pos+k} is a linear function of PE_{pos}

Proof sketch:

Using trigonometric identities:

$$\sin(\alpha + \beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$
$$\cos(\alpha + \beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

We can write:

$$[PE_{pos+k,2i} \ PE_{pos+k,2i+1}] = [\cos(\beta) \quad \sin(\beta) \ \ -\sin(\beta) \quad \cos(\beta)] \, [PE_{pos,2i} \ PE_{pos,2i+1}]$$

where $\beta = \frac{k}{10000^{2i/d}}$

Implication: Relative position k encoded by fixed transformation! Model can learn to attend by relative position.

[**DERIVATION NEEDED**: Full mathematical proof]

A3: Multi-Head Attention as Ensemble

Theorem: Multi-head attention with h heads learns h different attention patterns.

Intuition: Each head i has parameters (W_i^Q, W_i^K, W_i^V) that are learned independently.

The attention score for head i:

$$A_i = \text{softmax}\left(\frac{(XW_i^Q)(XW_i^K)^T}{\sqrt{d_k}}\right)(XW_i^V)$$

Final output combines all heads:

$$\text{Output} = \text{Concat}(A_1, \ldots, A_h)W^O$$

Each A_i can specialize to different patterns!

Empirical observation:

- Some heads track syntax
- Some heads track semantics
- Some heads track positional patterns
- Redundancy provides robustness

[END OF SLIDES]

Next Steps for Students

Before Next Class:

1. **Read**: "Attention Is All You Need" paper
2. **Code**: Implement scaled dot-product attention
3. **Experiment**: Train small Transformer on toy task
4. **Think**: How would you apply Transformers to your research area?

Lab Assignment:

Implement a mini-Transformer for character-level language modeling:

- Dataset: Shakespeare text
- Task: Predict next character
- Model: 4-layer decoder-only Transformer
- Deliverable: Working model + analysis

Project Ideas:

1. Compare Transformer vs RNN on long sequences
2. Visualize attention patterns in trained model
3. Implement efficient attention variant
4. Apply Transformer to new domain (graph, time series, etc.)
