An Algorithm for Routing Vectors in Sequences

Franz A. Heinsen
franz@glassroom.com

Abstract

We propose a routing algorithm that takes a sequence vectors and computes a new sequence with specified length and vector size. Each output vector maximizes "bang per bit," the difference between a net benefit to use and net cost to ignore data, by better predicting the input vectors. We describe output vectors as geometric objects, as latent variables that assign credit, as query states in a model of associative memory, and as agents in a model of a Society of Mind. We implement the algorithm with optimizations that reduce parameter count, computation, and memory use by orders of magnitude, enabling us to route sequences of greater length than previously possible. We evaluate our implementation on natural language and visual classification tasks, obtaining competitive or state-of-the-art accuracy and end-to-end credit assignments that are interpretable.[1]

1 Introduction

A longstanding goal in Artificial Intelligence is to formulate learning systems that assign credit, such that, when they succeed or fail in a task, we can determine and interpret which components of the system are responsible. A possible approach to the credit assignment problem is to route capsules at multiple levels of composition. A capsule is a group (*e.g.*, vector, matrix) of artificial neurons representing the properties of an entity in a context (*e.g.*, a token of text in a paragraph, an object depicted in an image). Routing consists of assigning data from input capsules, each representing a detected entity, to output capsules, each representing a detectable entity, by finding or computing agreement in some form (*e.g.*, identifying clusters) among candidate output capsules proposed by transforming the input capsules. Each output capsule is computed as a mixture of the candidates proposed for it on which the most input capsules agree, thereby assigning credit to those input capsules. If we compose multiple routings into a deep neural network, in every forward pass it assigns credit to input capsules representing the entities detected at each level of composition.

To date, deep neural networks applying various routing methods have shown promise in multiple domains, including vision and natural language, but only on small-scale tasks (Tsai et al., 2020) (Ribeiro et al., 2020) (Hahn et al., 2019) (Dou et al., 2019) (Heinsen, 2019) (Rajasegaran et al., 2019) (Xinyi and Chen, 2019) (Zhang et al., 2018) (Zhang et al., 2018) (Wang and Liu, 2018) (Hinton et al., 2018) (Sabour et al., 2017). Application of previously proposed routing methods to large-scale tasks has been impractical due to computational complexity, which increases in both space and time as a function of the length of input and output sequences, the number of elements per capsule, and the number of pairwise interactions between input, proposed, and output capsules.

Here, we adapt the routing algorithm proposed by Heinsen (2019) to operate on vectors as the capsules, generalize the algorithm by formulating it in terms of four neural networks (differentiable functions), and implement it with optimizations that reduce parameter count, memory use, and computation by orders of magnitude. The four neural networks are: $\mathcal{A}$ for obtaining an activation score per input vector, $\mathcal{F}$ for obtaining a different sequence of proposed output vectors given each input vector, $\mathcal{G}$ for predicting input vectors given a sequence of output vectors, and $\mathcal{S}$ for scoring

[1] Source code and instructions for replicating our results are online at https://github.com/glassroom/heinsen_routing.

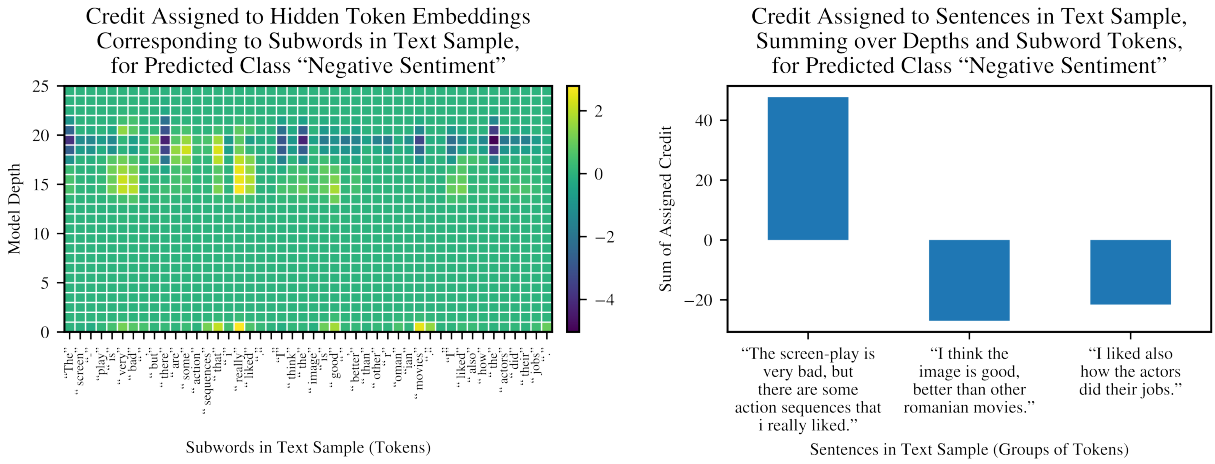

Figure 1: Typical example of end-to-end credit assignment, in this case for classifying the sentiment of a movie review. See Figures 8 and 9 for additional examples, including a typical example in vision.

actual versus predicted input vectors to quantify agreement. The algorithm is iterative. In each iteration, we update the state of all output vectors in parallel. We assign data from each input vector to the output vectors which best predict it, and compute each output vector's updated state by maximizing "bang per bit," the difference between a net benefit to use and a net cost to ignore input vector data. The output sequence's final state is that which maximizes "bang per bit" by best predicting, or explaining, the given input sequence.

Motivated by consilience, we describe output vectors from four different viewpoints: First, we describe them as geometric objects whose states are updated by linearly combining dynamically computed coefficients and components in a different basis for each output vector. Second, we describe output vectors as latent variables whose updated states are computed via credit assignments that are additive, like the Shapley values obtainable via SHAP methods (Lundberg and Lee, 2017), and also composable on their own (*i.e.*, independently of data transformations), subject to certain conditions. Third, we describe output vectors as query states in a model of associative memory, which, if we disregard the net cost to ignore data and restrict $\mathcal{A}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{S}$ in significant ways, reduces to a modern Hopfield network with the structure of a bipartite graph (Krotov and Hopfield, 2021) (Ramsauer et al., 2021), of which Transformer self-attention (Vaswani et al., 2017) is a notable special case. Fourth, we describe output vectors as agents competing to maximize utility by using or ignoring scarce resources through "knowledge lines," or K-lines, in a "block," which itself can interact with other blocks in a network modeling a Society of Mind (Minsky, 1986).

Our sample implementation of the algorithm incorporates three significant optimizations: First, we define $\mathcal{F}$ as the composition of a scaled tensor product and a linear transformation with position-wise biases, instead of as a different linear transformation and bias per interaction (as in the original variant of the algorithm), reducing parameter count by orders of magnitude. Second, we evaluate $\mathcal{F}$ lazily in each iteration, instead of eagerly before the first iteration, to avoid storing all elements of $\mathcal{F}$'s output simultaneously in memory as intermediate values, reducing memory footprint by orders of magnitude while increasing computation only linearly in the number of iterations. Third, we decompose the computation of all updated output vector states into a sequence of efficient tensor contractions, reducing memory footprint and computation by orders of magnitude.

We measure our implementation's parameter count, memory footprint, and execution time, and find they are linear in each of the number of input vectors, the size of input vectors, the number of output vectors, and the size of output vectors, enabling fine-grained control over memory consumption and computational cost. We successfully route input sequences with 1 million vectors, each a capsule with 1024 elements, at full (32-bit floating point) precision, keeping track of gradients, consuming under 18GB of memory on widely available commodity hardware. To the best of our knowledge, no implementation of any pre-

viously proposed routing method has been able to route as many capsules on any kind of hardware.

Finally, we evaluate our implementation on classification benchmarks in natural language and vision. In all benchmarks, we obtain accuracy competitive with or better than the state of the art, along with additive credit assignments that are composable independently of data transformations. We compute end-to-end credit assignments and find they are interpretable (Figures 1, 8, 9).

1.1 Notation

In mathematical expressions of tensor transformations, we show all indices as subscript text, implicitly assume broadcasting for any missing indices, perform all operations elementwise, and explicitly show all summations. Superscript text in parenthesis denotes labels. See Table 1 for examples. We do not use the notation of Linear Algebra because it cannot handle more than two indices. We do not use Einstein's implicit summation notation because it would require the use of operators for raising and lowering indices, adding complexity that is unnecessary for our purposes.

Example	Implementation in Python
$y_{ijk} \longleftarrow x^{(1)}_{ij} + x^{(2)}_{jk}$	`y = x1[:,:,None] + x2`
$y_{ijk} \longleftarrow x^{(1)}_{ij} x^{(2)}_{jk}$	`y = x1[:,:,None] * x2`
$y_{ik} \longleftarrow \sum_j x^{(1)}_{ij} x^{(2)}_{jk}$	`y = x1 @ x2`
$y_{ki} \longleftarrow e^{\sum_j x^{(1)}_{ij} x^{(2)}_{jk}}$	`y = (x1 @ x2).exp().T`
$y_{k} \longleftarrow \sum_{ij} x^{(1)}_{ij} x^{(2)}_{jk}$	`y = (x1 @ x2).sum(dim=0)`

Table 1: Examples of the notation we use, with all-subscript tensor indices, elementwise operations, implicit broadcasting, and explicit summations. In all examples, $x^{(1)}_{ij} \in \mathbb{R}^{d_1 \times d_2}$ and $x^{(2)}_{jk} \in \mathbb{R}^{d_2 \times d_3}$.

2 Proposed Routing Algorithm

The proposed algorithm executes a modified expectation-maximization loop with three steps: an E-Step for computing expected routing probabilities, a D-Step for computing shares of data used and ignored, and an M-Step for computing output vectors that maximize "bang per bit" by more accurately predicting the given input vectors. The original variant of the algorithm (Heinsen, 2019) routes matrices instead of vectors, in a loop with the same three steps, and computes output matrices as Gaussian mixtures that maximize the probability of generating the proposed ones, weighted by probabilities that maximize "bang per bit." *For ease of exposition, we describe the new variant of the algorithm assuming the reader has no familiarity with the original one.*

2.1 Overview

We show the proposed algorithm as Algorithm 1. Per sample, we accept a sequence of input vectors $x^{(\text{inp})}_{id}$ and return a sequence of output vectors $x^{(\text{out})}_{jh}$. The tensor indices, which we use consistently throughout the rest of this document, are:

$$\begin{aligned} i &= (1, 2, \ldots, n^{(\text{inp})}), \\ j &= (1, 2, \ldots, n^{(\text{out})}), \\ d &= (1, 2, \ldots, d^{(\text{inp})}), \\ h &= (1, 2, \ldots, d^{(\text{out})}), \end{aligned}$$

where $n^{(\text{inp})}$ and $n^{(\text{out})}$ are the number of input and output vectors, respectively, and $d^{(\text{inp})}$ and $d^{(\text{out})}$ are the size, or number of features, of input and output vectors, respectively.

In the following subsections, we walk through all steps of Algorithm 1 in order of execution.

2.2 Input Vector Activations

We apply a neural network $\mathcal{A}$ to the input vectors to obtain their activation scores $a^{(\text{inp})}_i$ (Algorithm 1, line 1), and subsequently apply a logistic function f to each activation score to obtain a probability per input vector $f(a^{(\text{inp})}_i)$ (Algorithm 1, lines 14-15).[2] We call each such probability an "input vector activation" and use it to gate the input vector's proposed output vectors.

A notable special case of $\mathcal{A}$, which we will revisit, is defining it as a constant function, $\mathcal{A}(\cdot) := \infty$, making all input vector activations $f(\infty) = 1$, in which case we always activate all (*i.e.*, never gate any) proposed output vectors.

2.3 Proposed Output Vectors, or Votes

We apply a neural network $\mathcal{F}$ to the input vectors to obtain a tensor of proposed output vectors V_{ijh} (Algorithm 1, line 2). The tensor V_{ijh} has, for each input vector i, a proposed vector for each possible output vector j with features h. We call each proposed output vector a "vote" to distinguish it from actual output vectors, which we compute at

[2] We represent the logistic function with f instead of σ, as is conventional, because the latter denotes standard deviation elsewhere in this document and in the original algorithm.

Algorithm 1: $\mathcal{A}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{S}$ are implementation-specific. f is the logistic function. $\beta_{ij}^{(\text{use})}$ and $\beta_{ij}^{(\text{ign})}$ are parameters if $n^{(\text{inp})}$ is fixed, implementation-specific transformations of $x_{id}^{(\text{inp})}$ otherwise.

Input: $x_{id}^{(\text{inp})} \in \mathbb{R}^{n^{(\text{inp})} \times d^{(\text{inp})}}$.
Output: $x_{jh}^{(\text{out})} \in \mathbb{R}^{n^{(\text{out})} \times d^{(\text{out})}}$.

1 $a_i^{(\text{inp})} \longleftarrow \mathcal{A}(x_{id}^{(\text{inp})}), \quad \mathcal{A} : \mathbb{R}^{n^{(\text{inp})} \times d^{(\text{inp})}} \to \mathbb{R}^{n^{(\text{inp})}}$ // obtain input vector activation scores
2 $V_{ijh} \longleftarrow \mathcal{F}\left(x_{id}^{(\text{inp})}\right), \quad \mathcal{F} : \mathbb{R}^{n^{(\text{inp})} \times d^{(\text{inp})}} \to \mathbb{R}^{n^{(\text{inp})} \times n^{(\text{out})} \times d^{(\text{out})}}$ // obtain votes (proposed output vectors)
3 **for** $n^{(\text{iters})}$ *iterations* **do**
4 **begin** E-Step
5 **if** *on first iteration* **then**
6 $R_{ij} \longleftarrow \frac{1}{n^{(\text{out})}}$ // assign flat prior in first iteration
7 **else**
8 $\hat{x}_{jd}^{(\text{inp})} \longleftarrow \mathcal{G}(x_{jh}^{(\text{out})}), \quad \mathcal{G} : \mathbb{R}^{n^{(\text{out})} \times d^{(\text{out})}} \to \mathbb{R}^{n^{(\text{out})} \times d^{(\text{inp})}}$ // predict input vectors
9 $S_{ij} \longleftarrow \mathcal{S}(x_{id}^{(\text{inp})}, \hat{x}_{jd}^{(\text{inp})}), \quad \mathcal{S} : \mathbb{R}^{n^{(\text{inp})} \times d^{(\text{inp})}} \times \mathbb{R}^{n^{(\text{out})} \times d^{(\text{inp})}} \to \mathbb{R}^{n^{(\text{inp})} \times n^{(\text{out})}}$ // score the predictions
10 $R_{ij} \longleftarrow \frac{e^{S_{ij}}}{\sum_j e^{S_{ij}}}$ // normalize to distributions
11 **end**
12 **end**
13 **begin** D-Step
14 $D_{ij}^{(\text{use})} \longleftarrow f(a_i^{(\text{inp})}) R_{ij}$ // compute shares of data used
15 $D_{ij}^{(\text{ign})} \longleftarrow f(a_i^{(\text{inp})}) - D_{ij}^{(\text{use})}$ // compute shares of data ignored
16 **end**
17 **begin** M-Step
18 $x_{jh}^{(\text{out})} \longleftarrow \sum_i \beta_{ij}^{(\text{use})} D_{ij}^{(\text{use})} V_{ijh} - \sum_i \beta_{ij}^{(\text{ign})} D_{ij}^{(\text{ign})} V_{ijh}$ // maximize "bang per bit"
19 **end**
20 **end**

the end of each iteration in the routing loop. $\mathcal{F}$ *should break symmetry*, *i.e.*, obtain from each input vector a different vote for each output vector; otherwise, all votes from each input vector i would be identical and routing would be pointless.

A notable special case of $\mathcal{F}$, which we will revisit, is defining it as a constant function that returns a parameter, $\mathcal{F}(\cdot) := W_{ijh}^{(\text{mem})}$, making all votes "learnable memories" that are independent of the given input vectors, retrieved instead of computed from them at inference.

Implementing $\mathcal{F}$ presents two difficulties to routing long sequences. First, storing the votes V_{ijh} requires $\mathcal{O}(n^{(\text{inp})} n^{(\text{out})} d^{(\text{out})})$ space, which becomes impractical as we increase the length of input and output sequences. Second, naive approaches to breaking symmetry require parameter counts that also become impractical as we increase the length of input and output sequences. For example, applying $n^{(\text{inp})} n^{(\text{out})}$ different linear transformations (as in the original variant of the algorithm) would require $n^{(\text{inp})} n^{(\text{out})} d^{(\text{inp})} d^{(\text{out})}$ parameters, and $n^{(\text{out})}$ linear transformations would require $n^{(\text{out})} d^{(\text{inp})} d^{(\text{out})}$ parameters. In Section 4, we present a sample implementation with optimizations that overcome both difficulties, by lazily computing, weighting, and contracting V_{ijh}'s elements without storing all of them simultaneously as intermediate values, in an efficient manner. For now, we set aside concerns about routing longer sequences and focus on the next step of the algorithm: the routing loop.

2.4 Routing Loop

2.4.1 E-Step

The E-Step computes a tensor R_{ij} of expected routing probabilities, for assigning data from each input vector i's votes to compute each output vector j. In the first iteration, we assign equal routing probability, $\frac{1}{n^{(\text{out})}}$, *i.e.*, a flat prior, over the votes from each input vector (Algorithm 1, line 6).

In subsequent iterations, we assign greater routing probability to the output vector states which best predict the input vectors, as follows: First, we predict input vectors by applying a neural network $\mathcal{G}$ to the the previous iteration's output vector states (line 8). We obtain $n^{(\text{out})}$ predicted input vectors. Second, we compute prediction scores S_{ij} by applying a neural network $\mathcal{S}$ to actual and predicted input vectors (line 9). $\mathcal{S}$ may compute a symmetric kernel (dot-product, Euclidean distance, radial basis function, etc.) or a non-

symmetric kernel (*i.e.*, one that computes scores differently for different pairs of actual and predicted input vectors, breaking symmetry over the input vectors too).[3] Finally, we apply a Softmax function to S_{ij}, normalizing over index j, to obtain updated routing probabilities R_{ij} which add up to 1 per input vector; *i.e.*, for each input vector i we obtain a distribution over the input vector's proposed output vector states j (line 10).

2.4.2 D-Step

The D-Step computes the shares of data used $D_{ij}^{(\text{use})}$ and ignored $D_{ij}^{(\text{ign})}$ from each input vector i's vote for computing the state of each output vector j. We use these shares to put output vectors in competition with each other as they try to use "more valuable bits" and ignore "less valuable bits" of data from each input vector's votes, such that each output vector can use more data from an input vector's votes only if all other output vectors collectively ignore it, and vice versa.

We obtain $D_{ij}^{(\text{use})}$ by multiplying input vector activations $f(a_i^{(\text{inp})})$ by routing probabilities R_{ij} (line 14). Each element of $f(a_i^{(\text{inp})})$ is in $[0, 1]$ and the elements of R_{ij} along index j add up to 1, so the elements of $D_{ij}^{(\text{use})}$ have values that range from 0 ("ignore all data from input vector i's vote for output vector j") to 1 ("use all data from input vector i's vote for output vector j"), but never exceed each input vector activation ("how much data from input vector i's votes can all output vectors collectively use?"). We then compute $D_{ij}^{(\text{ign})}$ by subtracting the shares used from the input vector activations (line 15), such that for every input vector i and every output vector j

$$\begin{aligned} D_{ij}^{(\text{use})} + D_{ij}^{(\text{ign})} &= f(a_i^{(\text{inp})}) \\ \sum_j D_{ij}^{(\text{use})} &= f(a_i^{(\text{inp})}), \end{aligned} \tag{1}$$

where

$$\begin{aligned} 0 \le D_{ij}^{(\text{use})} \le f(a_i^{(\text{inp})}) \le 1 \\ 0 \le D_{ij}^{(\text{ign})} \le f(a_i^{(\text{inp})}) \le 1, \end{aligned} \tag{2}$$

treating activated (non-gated) data as a scarce resource that cannot be wasted: Every bit must be "fully used" by one or more output vectors and "fully ignored" by all other output vectors.

[3] Optionally, $\mathcal{G}$ may specify a generative model that samples the predicted input vectors given current output vector states, in which case $\mathcal{S}$ should compute or approximate the conditional log-probability densities of actual input vectors, given the predicted input vectors, as the scores S_{ij}.

2.4.3 M-Step

The M-Step computes updated output vector states $x_{jh}^{(\text{out})}$ at the end of each iteration as the difference between each output vector's net benefit to use and net cost to ignore data from input vector votes, maximizing "bang per bit" (line 18). The word "net" denotes that values may be positive or negative—*i.e.*, it is possible for the net benefit to be negative and for the net cost to be positive.

We compute each output vector's net benefit to use data as a linear combination of the votes, where the coefficients are the shares of data used, scaled by a parameter $\beta_{ij}^{(\text{use})}$ quantifying each output vector's net benefit per unit of data to use each vote—hence the term "bang per bit." We compute the net cost to ignore data also as a linear combination of the votes, where the coefficients are the shares of data ignored, scaled by a parameter $\beta_{ij}^{(\text{ign})}$ quantifying each output vector's net cost per unit of data to ignore each vote.

For example, the first output vector's state is

$$\begin{aligned} x_{1h}^{(\text{out})} \longleftarrow \quad & \sum_i \beta_{i1}^{(\text{use})} D_{i1}^{(\text{use})} V_{i1h} \quad \text{// net benefit} \\ & - \sum_i \beta_{i1}^{(\text{ign})} D_{i1}^{(\text{ign})} V_{i1h} \quad \text{// net cost} \end{aligned} \tag{3}$$

where the tensor slice V_{i1h} has the votes from input vectors i for output vector 1 with elements h, $D_{i1}^{(\text{use})}$ and $D_{i1}^{(\text{ign})}$ are the shares of data from each input vector i used and ignored by output vector 1, and $\beta_{i1}^{(\text{use})}$ and $\beta_{i1}^{(\text{ign})}$ are the net benefit and net cost per unit of data from input vector i for output vector 1. We maximize the first output vector's net benefit from those votes it uses, less its net cost from those votes it ignores, in competition with all other output vectors, for which we do the same.

If no output vector can improve its net benefit less net cost, given the state of all other output vectors, the routing loop has reached a local optimum in a "bang per bit" landscape, specific to the implementation of neural networks $\mathcal{A}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{S}$, given the current sequence of input vectors.

2.5 Training

We optimize all parameters for a training objective specified elsewhere as a dependency of the output vector states, which in turn are a function of (a) $\beta_{ij}^{(\text{use})}$ and $\beta_{ij}^{(\text{ign})}$, (b) $D_{ij}^{(\text{use})}$ and $D_{ij}^{(\text{ign})}$, and (c) V_{ijh} in each iteration. Provided the implementation of $\mathcal{A}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{S}$ exhibits Lyapunov stability in the routing loop, we can directly optimize

(a), which are learnable parameters, and (c), the votes, which are proposed by a differentiable function ($\mathcal{F}$), but not (b), which we can optimize only indirectly, via the interaction of input vector activations $f(a_i^{(\text{inp})})$ and actual-versus-predicted input vector scores S_{ij}, which together determine $D_{ij}^{(\text{use})}$ and $D_{ij}^{(\text{ign})}$, subject to (1) and (2), *inducing the algorithm to learn to activate and predict input vectors* as we optimize for the training objective.

If the training objective induces each output vector's elements to represent the properties of an object, concept, relationship, or other entity for which we, human beings, already have a label, each output vector is *a symbol for a known entity*. Otherwise, the algorithm learns to compute output vector states representing objects, concepts, relationships, or other entities for which we, human beings, may or may not have labels (*e.g.*, we may be unaware of their existence), making each output vector *a symbol for a discoverable entity*.

3 Understanding Output Vectors

3.1 As Geometric Objects

If we factorize out V_{ijh} from the expression in line 18 of Algorithm 1, we see that each iteration computes the updated state of each output vector as the linear combination of a vector basis in V_{ijh} with corresponding "bang per bit" coefficients ϕ_{ij}:

$$x_{jh}^{(\text{out})} \longleftarrow \sum_i \underbrace{(\beta_{ij}^{(\text{use})} D_{ij}^{(\text{use})} - \beta_{ij}^{(\text{ign})} D_{ij}^{(\text{ign})})}_{\text{Define as } \phi_{ij}} V_{ijh}$$
$$= \sum_i \underbrace{\phi_{ij}}_{\substack{\text{Coeffi-}\\\text{cients}}} \underbrace{V_{ijh}}_{\substack{\text{Vector}\\\text{bases}}}. \tag{4}$$

Neural network $\mathcal{F}$ transforms input vectors into *a different basis for each output vector*. The tensor V_{ijh} consists of $n^{(\text{inp})}$ votes specifying a basis for each of $n^{(\text{out})}$ output vectors of size $d^{(\text{out})}$.[4] In the special case where $\mathcal{F}$ is a constant function that returns a parameter $W_{ijh}^{(\text{mem})}$, every basis is a learned memory, retrieved given the input vectors instead of computed from them at inference. Each basis may represent a different feature space.

For example, the computation of the first output vector's state $x_{1h}^{(\text{out})}$ in (3) is factorized as

$$x_{1h}^{(\text{out})} \longleftarrow \sum_i \phi_{i1} V_{i1h} \tag{5}$$

where the tensor slice V_{i1h} is the basis specified by votes i for output vector 1 with elements h, and tensor slice ϕ_{i1} has the coefficients i for the votes that specify output vector 1's basis. Figure 2 illustrates an example with three bases $(V_{i1h}, V_{i2h}, V_{i3h})$ and three slices with coefficients $(\phi_{i1}, \phi_{i2}, \phi_{i3})$ obtained from two input vectors for computing the state of three output vectors.

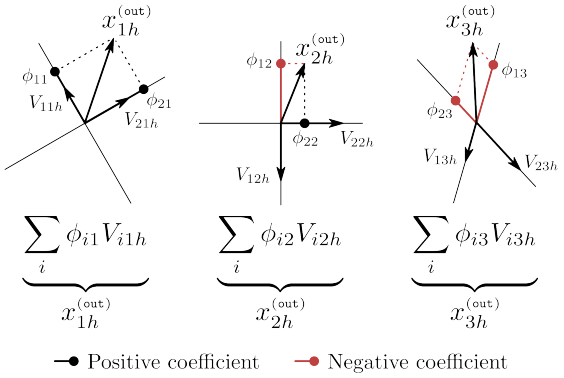

Figure 2: Each output vector j's state is the linear combination of its corresponding basis ih in V_{ijh} with its dynamically updated coefficients i in ϕ_{ij}. In this illustration, $n^{(\text{inp})} = 2$ and $n^{(\text{out})} = 3$.

When we maximize "bang per bit," we find the coordinates in each basis that best predict the input vectors, subject to constraints (1) and (2), in service of a training objective, specified elsewhere as a dependency of the final output vector states.

3.2 As Latent Variables that Assign Credit

We can describe each output vector as a latent or explanatory variable with $d^{(\text{out})}$ elements. From this viewpoint, each basis in V_{ijh} is a space of "proposed hypotheses" for one output vector. Different coordinates in each basis represent different hypotheses for explaining, or predicting, the given sequence of input vectors (Figure 3). In the special case where $\mathcal{F}$ is a constant function that returns a parameter $W_{ijh}^{(\text{mem})}$, every space of proposed hypotheses is a learned memory, retrieved instead of computed from the input vectors at run time.

The "bang per bit" coefficients ϕ_{ij} (4), or coordinates in each space of proposed hypotheses, specify how much each input vector i's proposed hypothesis adds to, or subtracts from, each output

[4] For intuition's sake, we can think of each vote as a basis vector, even though the vote is properly a basis vector only if it is linearly independent of all other votes in the same basis.

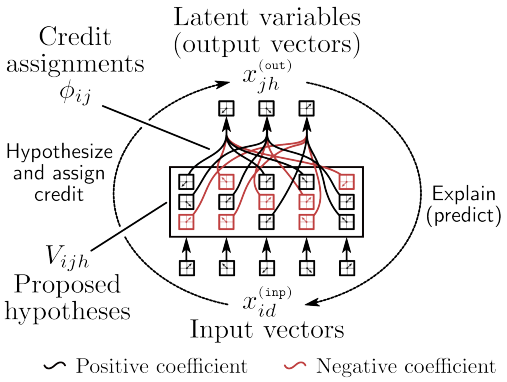

Figure 3: We find the credit assignments in each space of proposed hypotheses that compute the output vector states which best explain the input vectors. In this diagram, $n^{(\text{inp})} = 5$ and $n^{(\text{out})} = 3$.

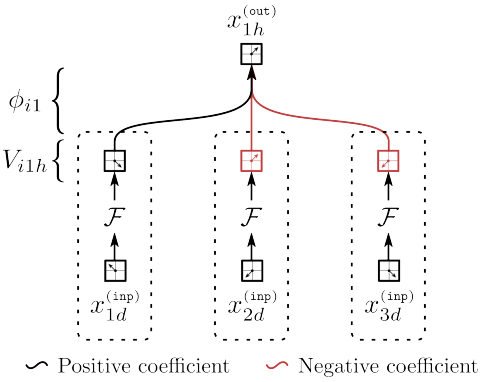

Figure 4: If $\mathcal{F}$ keeps data from each input vector separate, then data from different input vectors is mixed only by ϕ_{ij}. Here, we show three independently obtained votes for the first output vector.

vector j's updated state. That is, the coefficients *assign credit via addition and subtraction* of each input vector i's proposed hypothesis to compute each output vector j's updated state. Compared to SHAP methods (Lundberg and Lee, 2017), which estimate additive credit assignments by sampling model outputs on a sufficiently large number of perturbations applied to a given input sample, our algorithm gives us additive credit assignments "for free" via an iterative forward pass, without having to figure out how best to perturb input data.

From this viewpoint, maximizing "bang per bit" means finding the credit assignments ϕ_{ij} in all spaces of proposed hypotheses V_{ijh}, for computing the output vector states $x_{jh}^{(\text{out})}$ which best explain the sequence of input vectors $x_{id}^{(\text{inp})}$, in service of a training objective specified elsewhere as a dependency of the final output vector states. If no output vector can improve its predictions, given the state of all other output vectors, the algorithm has reached a local credit-assignment optimum in a landscape of proposed hypotheses specific to the implementation of $\mathcal{A}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{S}$.

If we implement $\mathcal{F}$ to obtain each input vector's votes independently of other input vectors' votes, then data from different input vectors is mixed only by ϕ_{ij} (Figure 4), making the credit assignments composable on their own, independently of data transformations: In a network of routings, data from different vectors is mixed only by the final credit assignments computed by each routing. In appendix A, we show methods for computing end-to-end credit assignments over common compositions of routings, including residual layers.

3.3 As Associative Memory Query States

We can describe the proposed algorithm as applying an update rule $\mathcal{U}$ to output vectors in each iteration, given a sequence of input vectors:

$$\underbrace{x_{jh}^{(\text{out})}}_{\text{Updated}} \longleftarrow \mathcal{U}(x_{jh}^{(\text{out})} | x_{id}^{(\text{inp})}), \tag{6}$$

where $\mathcal{U}$ composes all transformations we apply to output vectors in the E-Step, D-Step, and M-Step after the first iteration (lines 8-18). Grouping all such transformations into three newly defined neural networks, which we call $\mathcal{R}$, $\mathcal{M}$, and $\mathcal{B}$,[5] we see that $\mathcal{U}$ is the update rule for a model of associative memory with the structure of bipartite graph in which output vectors are query states and input vectors are keys to content-addressable "memory values" and "memory biases:"

$$\mathcal{U}(x_{jh}^{(\text{out})} | x_{id}^{(\text{inp})}) := \sum_i \Big(\mathcal{R}(\underbrace{x_{jh}^{(\text{out})}}_{\text{Queries}} | \underbrace{x_{id}^{(\text{inp})}}_{\text{Keys}}) \underbrace{\mathcal{M}(x_{id}^{(\text{inp})})}_{\text{Values}} - \underbrace{\mathcal{B}(x_{id}^{(\text{inp})})}_{\text{Biases}} \Big), \tag{7}$$

where $\mathcal{R}$ applies $\mathcal{G}$ and $\mathcal{S}$ to obtain updated routing probabilities R_{ij} (E-Step, lines 8-10),

$$\mathcal{R}(x_{jh}^{(\text{out})} \mid x_{id}^{(\text{inp})}) := \frac{e^{\mathcal{S}(x_{id}^{(\text{inp})}, \mathcal{G}(x_{jh}^{(\text{out})}))}}{\sum_j e^{\mathcal{S}(x_{id}^{(\text{inp})}, \mathcal{G}(x_{jh}^{(\text{out})}))}}, \tag{8}$$

[5] See appendix B for the derivation of $\mathcal{U}$ in terms of these three newly defined neural networks: $\mathcal{R}$, $\mathcal{M}$, and $\mathcal{B}$.

and $\mathcal{M}$ and $\mathcal{B}$ compose and weight the application of $\mathcal{A}$ and $\mathcal{F}$ in the D-Step and M-Step to obtain memory values and biases for each key,

$$\begin{aligned}\mathcal{M}(x_{id}^{(\text{inp})}) &:= (\beta_{ij}^{(\text{use})} + \beta_{ij}^{(\text{ign})}) f(\mathcal{A}(x_{id}^{(\text{inp})}))\mathcal{F}(x_{id}^{(\text{inp})}) \\ \mathcal{B}(x_{id}^{(\text{inp})}) &:= \beta_{ij}^{(\text{ign})} f(\mathcal{A}(x_{id}^{(\text{inp})}))\mathcal{F}(x_{id}^{(\text{inp})}),\end{aligned} \tag{9}$$

i.e., $\mathcal{M}$ and $\mathcal{B}$ compute different scalings of the input vector votes, $\mathcal{F}(x_{id}^{(\text{inp})})$, gated by corresponding input vector activations, $f(\mathcal{A}(x_{id}^{(\text{inp})}))$.

In the special case where $\mathcal{F}$ is a constant function that returns a parameter $W_{ijh}^{(\text{mem})}$, the votes are learned memories, retrieved instead of computed from the keys, and $\mathcal{M}$ and $\mathcal{B}$ compute different gated scalings of such retrieved memories. If $\mathcal{A}$ is a constant function that returns ∞, all memories are always fully activated (*i.e.*, never gated).

The initial query states assign equal prior routing probability, $\frac{1}{n^{(\text{out})}}$ (E-Step, line 6), to their corresponding memory values given each key:

$$\underbrace{x_{jh}^{(\text{out})}}_{\text{Initial}} \longleftarrow \sum_i \Big(\underbrace{\tfrac{1}{n^{(\text{out})}}}_{\text{Prior}} \underbrace{\mathcal{M}(x_{id}^{(\text{inp})})}_{\text{Values}} - \underbrace{\mathcal{B}(x_{id}^{(\text{inp})})}_{\text{Biases}} \Big). \tag{10}$$

When we maximize "bang per bit," we iteratively update query states as the mixtures of memory values, less memory biases, which best predict the given keys, in service of a training objective that we specify elsewhere as a dependency of the final query states. If no query can improve its predictions, given the state of all other queries, we have reached a local maximum in a "bang per bit" landscape (or equivalently, a local minimum in an energy landscape) specific to the implementation of neural networks $\mathcal{A}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{S}$.

Provided the implementation exhibits Lyapunov stability, we can view the algorithm as an "infinitely deep" recurrent neural network that repeatedly applies the same layer $\mathcal{U}$ to the queries until they converge to a stable state $\breve{x}_{jh}^{(\text{out})}$:

$$\begin{aligned}\breve{x}_{jh}^{(\text{out})} &= \lim_{t\to\infty} \mathcal{U}^t(x_{jh}^{(\text{out})} | x_{id}^{(\text{inp})}) \\ &= \mathcal{U}(\breve{x}_{jh}^{(\text{out})} | x_{id}^{(\text{inp})}),\end{aligned} \tag{11}$$

where $\mathcal{U}^t(x_{jh}^{(\text{out})} | x_{id}^{(\text{inp})})$ denotes t applications of $\mathcal{U}$ to the queries, given the keys. Alternatively, we can think of the algorithm as a "single layer" implicitly defined by its output, the stable state $\breve{x}_{jh}^{(\text{out})}$ that solves (11), making the algorithm a "deep equilibrium model" (Bai et al., 2019).[6]

We believe our algorithm is the first model of associative memory to take into account a net cost to ignore data. If we simplify the algorithm, it reduces to a modern Hopfield network with bipartite structure (Krotov and Hopfield, 2021) (Ramsauer et al., 2021), of which Transformer self-attention (Vaswani et al., 2017) is a notable special case. The necessary simplifications are: (a) We would have to disregard the net cost to ignore data, *e.g.*, by restricting $\beta_{ij}^{(\text{ign})}$ to constant 0, eliminating the memory biases obtained by $\mathcal{B}$ from expressions (7) and (10). (b) We would have to restrict $\beta_{ij}^{(\text{use})}$ to constant 1, so as to avoid scaling votes differently for each pair of input and output vectors. (c) We would have to restrict $\mathcal{A}$ to a constant function that returns ∞, always fully activating all (*i.e.*, never gating any) memory values obtained by $\mathcal{M}$. (d) We would have to restrict $\mathcal{F}$ (and thus $\mathcal{M}$) to propose only one sequence of proposed output vectors, *i.e.*, not to break symmetry over them, making routing unnecessary, and apply instead attention over that single sequence of proposed output vectors. (e) We would have to restrict $\mathcal{G}$ to the transformations considered by Krotov and Hopfield (2021) and Ramsauer et al. (2021). (f) We would have to restrict $\mathcal{S}$ to apply the same symmetric kernel function (multi-head dot-product) to all pairs of actual and predicted input vectors.

3.4 As Agents in a Society of Mind

Output vectors are multidimensional agents competing with each other to use or ignore data representing input vectors. Each input vector is a scarce resource that cannot be wasted, as we account for all data, ensuring each agent can use or ignore it only at the expense of other agents, as described in 2.4. Agents improve their use and ignore shares by more accurately predicting the scarce resources.

Neural network $\mathcal{F}$ transforms each scarce resource, or input vector, into a different represen-

[6] We consider only query states that evolve over a discrete number of iterations. Were we to extend our algorithm to the continuous setting, query states would evolve instead over time t by a system of ordinary differential equations:

$$\frac{\partial}{\partial t} x_{jh}^{(\text{out})}(t) = \mathcal{U}'\left(x_{jh}^{(\text{out})}(t) \,|\, x_{id}^{(\text{inp})}\right),$$

with initial condition at $t = t_0$ given by an uniform prior distribution over each query's corresponding memory values given each key (10). Alas, absent an analytical solution (or plausible implementation as a continuous physical process), we would have to approximate integration with numerical methods, requiring a discrete number of iterations anyway.

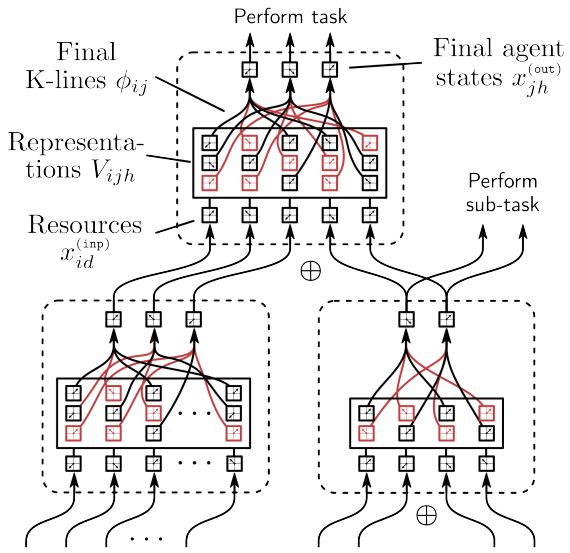

Figure 5: A network of blocks in a model of a Society of Mind (Minsky, 1986). In each block, agents use or ignore representations of resources via K-lines to perform tasks learned in training.

tation for each agent. In the special case where $\mathcal{F}$ is a constant function that returns a parameter $W^{(\text{mem})}_{ijh}$ with learned memories, agents compete against each other to use or ignore, not representations computed from the actual scarce resources, but representations learned independently of such resources—"imagined resources," as it were.

From this viewpoint, "bang per bit" is a form of *utility*, and parameters $\beta^{(\text{use})}_{ij}$ and $\beta^{(\text{ign})}_{ij}$ are *net prices* each agent pays or collects per unit of data used or ignored to maximize utility, in service of a training objective specified elsewhere as a dependency of the final agent states. If no agent can improve its utility, given the state of all other agents, the game has reached a local utility optimum specific to the implementation of $\mathcal{A}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{S}$.

The "bang per bit" coefficients ϕ_{ij} (4) function as "knowledge lines," or *K-lines*, connecting agents to representations of resources as necessary to perform tasks learned in training. If we call an instance of the algorithm a "block," multiple blocks can interact with each other via their respective agents' final states, dynamically connected via K-lines to perform tasks learned in training, modeling a Society of Mind (Minsky, 1986), as shown in Figure 5, but with one significant difference: In our algorithm, the agents in each block incur a net cost for ignoring their representations of the available resources.

4 Efficient Implementation

The number of possible implementations of $\mathcal{A}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{S}$ is infinite. Here, we present one implementation (Algorithm 2), incorporating three significant optimizations that reduce parameter count, memory use, and computation by orders of magnitude, overcoming the difficulties to routing longer sequences discussed in 2.3.

4.1 Efficient Implementation of $\mathcal{F}$

Our first significant optimization is to implement $\mathcal{F}$ with orders of magnitude fewer parameters than would be necessary were we to apply a different set of linear transformations per output vector, as described in 2.3.

We define $\mathcal{F}$ as a two-layer neural network:

$$\mathcal{F}(\cdot) := \mathcal{F}_2(\mathcal{F}_1(\cdot)), \tag{12}$$

where

$$\begin{aligned} \mathcal{F}_1(\cdot) &:= \frac{1}{\sqrt{n^{(\text{inp})}}}(\cdot)W^{(\mathcal{F}_1)}_{jd} \\ \mathcal{F}_2(\cdot) &:= \sum_d W^{(\mathcal{F}_2)}_{dh}(\cdot) + B^{(\mathcal{F}_2)}_{jh}. \end{aligned} \tag{13}$$

When we apply $\mathcal{F}$ to a sequence of input vectors $x^{(\text{inp})}_{id}$, $\mathcal{F}_1$ computes a tensor product:

$$\underbrace{\mathbb{R}^{n^{(\text{inp})}\times d^{(\text{inp})}}}_{x^{(\text{inp})}_{id}} \otimes \underbrace{\mathbb{R}^{n^{(\text{out})}\times d^{(\text{inp})}}}_{W^{(\mathcal{F}_1)}_{jd}} \to \underbrace{\mathbb{R}^{n^{(\text{inp})}\times n^{(\text{out})}\times d^{(\text{inp})}}}_{x^{(\text{inp})}_{id}W^{(\mathcal{F}_1)}_{jd}}, \tag{14}$$

giving us, for each of $n^{(\text{inp})}$ input vectors, $n^{(\text{out})}$ different elementwise scalings, or Hadamard products, of its $d^{(\text{inp})}$ elements. We scale the tensor product by $\frac{1}{\sqrt{n^{(\text{inp})}}}$ to keep the subsequent contraction of votes over index i in the same region for different values of $n^{(\text{inp})}$. $\mathcal{F}_2$ applies parameter $W^{(\mathcal{F}_2)}_{dh}$ as a linear transformation from $\mathbb{R}^{d^{(\text{inp})}}$ to $\mathbb{R}^{d^{(\text{out})}}$, and then adds $B^{(\mathcal{F}_2)}_{jh}$, a different bias per output vector basis, making it possible for all bases to span up to $d^{(\text{out})}$ dimensions when $n^{(\text{inp})} \geq d^{(\text{out})}$ but $d^{(\text{inp})} < d^{(\text{out})}$.[7] The tensor product and per-basis biases break symmetry.

$\mathcal{F}$'s parameter count in this implementation is $n^{(\text{out})}d^{(\text{inp})} + d^{(\text{inp})}d^{(\text{out})} + n^{(\text{out})}d^{(\text{out})}$, versus $n^{(\text{inp})}n^{(\text{out})}d^{(\text{inp})}d^{(\text{out})}$ were we to apply $n^{(\text{inp})}n^{(\text{out})}$ different linear transformations, or $n^{(\text{out})}d^{(\text{inp})}d^{(\text{out})}$

[7] If $d^{(\text{inp})} < d^{(\text{out})}$ and we don't add a different bias to each basis, all bases would span the same subspace of dimension $n \leq \min\left(n^{(\text{inp})}, \text{rank}(W^{(\mathcal{F}_2)}_{dh})\right) < d^{(\text{out})}$.

Algorithm 2: Our implementation of $\mathcal{A}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{S}$. Trivial optimizations are not shown for ease of exposition. $\mathfrak{N}$ denotes normalization of each vector's elements to zero mean and unit variance for numerical stability. If $n^{(\text{inp})}$ is variable, we remove index i from all parameters that have it and compute $\beta^{(\text{use})}_{ij} \longleftarrow \sum_d x^{(\text{inp})}_{id} W^{(\text{use})}_{dj} + B^{(\text{use})}_j$ and $\beta^{(\text{ign})}_{ij} \longleftarrow \sum_d x^{(\text{inp})}_{id} W^{(\text{ign})}_{dj} + B^{(\text{ign})}_j$.

Input: $x^{(\text{inp})}_{id}$.
Output: $x^{(\text{out})}_{jh}$.

1 $a^{(\text{inp})}_i \longleftarrow \frac{\sum_d W^{(\mathcal{A})}_{id} x^{(\text{inp})}_{id}}{\sqrt{n^{(\text{inp})}}} + B^{(\mathcal{A})}_i$ // apply $\mathcal{A}$, a scaled linear transformation with bias per input vector i
2 **for** $n^{(\text{iters})}$ *iterations* **do**
3 **begin** E-Step
4 **if** *on first iteration* **then**
5 $R_{ij} \longleftarrow \frac{1}{n^{(\text{out})}}$
6 **else**
7 $\hat{x}^{(\text{inp})}_{jd} \longleftarrow W^{(\mathcal{G}_2)}_{jd} \sum_h W^{(\mathcal{G}_1)}_{hd} \mathfrak{N}(x^{(\text{out})}_{jh}) + B^{(\mathcal{G}_2)}_{jd}$ // apply $\mathcal{G}$, a two-layer neural network per output vector j
8 $S_{ij} \longleftarrow \log f\left(W^{(\mathcal{S})}_{ij} \sum_d x^{(\text{inp})}_{id} \hat{x}^{(\text{inp})}_{jd} + B^{(\mathcal{S})}_{ij}\right)$ // apply $\mathcal{S}$, a nonlinear transformation per dot-product ij
9 $R_{ij} \longleftarrow \frac{e^{S_{ij}}}{\sum_j e^{S_{ij}}}$
10 **end**
11 **end**
12 **begin** D-Step
13 $D^{(\text{use})}_{ij} \longleftarrow f(a^{(\text{inp})}_i) R_{ij}$
14 $D^{(\text{ign})}_{ij} \longleftarrow f(a^{(\text{inp})}_i) - D^{(\text{use})}_{ij}$
15 **end**
16 **begin** M-Step
17 $\phi_{ij} \longleftarrow \beta^{(\text{use})}_{ij} D^{(\text{use})}_{ij} - \beta^{(\text{ign})}_{ij} D^{(\text{ign})}_{ij}$ // compute "bang per bit" coefficients i for each basis j
18 $x^{(\text{out})}_{jh} \longleftarrow \frac{\sum_d W^{(\mathcal{F}_2)}_{dh} W^{(\mathcal{F}_1)}_{jd} \sum_i \phi_{ij} x^{(\text{inp})}_{id}}{\sqrt{n^{(\text{inp})}}} + \sum_i \phi_{ij} B^{(\mathcal{F}_2)}_{jh}$ // lazily evaluate $\mathcal{F}\left(x^{(\text{inp})}_{id}\right)$ and efficiently contract votes
19 **end**
20 **end**

were we to apply $n^{(\text{out})}$ different linear transformations to each input vector. The trade-off of this reduction in parameter count is that, for any fixed $n^{(\text{inp})}$, $n^{(\text{out})}$, $d^{(\text{inp})}$, and $d^{(\text{out})}$, the space of transformations learnable by $\mathcal{F}$ is smaller.

4.2 Lazy Evaluation of $\mathcal{F}$

Our second significant optimization is to evaluate $\mathcal{F}$ lazily in each iteration, in order to compute and contract votes as needed without having to store all of them simultaneously in memory as intermediate values: The tensor V_{ijh} disappears from all expressions. Only the output vectors need be stored at the end of each iteration (Algorithm 2, line 18). The lazy evaluation of $\mathcal{F}$ and immediate contraction of each vote can be done efficiently, *i.e.*, in parallel, because our implementation of $\mathcal{F}$ computes each input vector's vote for each output vector independently from every other vote.

By never storing votes in a tensor V_{ijh}, we reduce memory footprint by $\mathcal{O}(n^{(\text{inp})} n^{(\text{out})} d^{(\text{out})})$. The trade-off of this reduction in memory is an increase in computation that is linear in the number of iterations: We now compute all votes in every iteration, instead of only once before the loop.

4.3 Efficient Evaluation of $\mathcal{F}$

Our third significant optimization is necessary to avoid having to store intermediate-value tensors with $n^{(\text{inp})} \times n^{(\text{out})} \times d^{(\text{inp})}$ or $n^{(\text{inp})} \times n^{(\text{out})} \times d^{(\text{out})}$ elements simultaneously in memory, and also to avoid computing votes twice in each iteration, which is a side effect of the lazy evaluation of $\mathcal{F}$, due to our computation of output vectors as a difference of two weighted sums of votes (Algorithm 1, line 18), both now lazily evaluated.

We factorize the difference of weighted sums into the tensor contraction $\sum_i \phi_{ij} \mathcal{F}_2(\mathcal{F}_1(x^{(\text{inp})}_{id}))$, where ϕ_{ij} are the "bang per bit" coefficients (Algorithm 2, line 17), and algebraically manipulate it to obtain the expression in Algorithm 2, line 18. The expression computes, weights, and contracts votes in a memory-efficient manner in each iteration, and then applies $\mathcal{F}_2$ as a last step, *after* contracting all votes, reducing the number of linear transformations executed in parallel by a factor of $n^{(\text{inp})}$. See appendix C for the derivation.

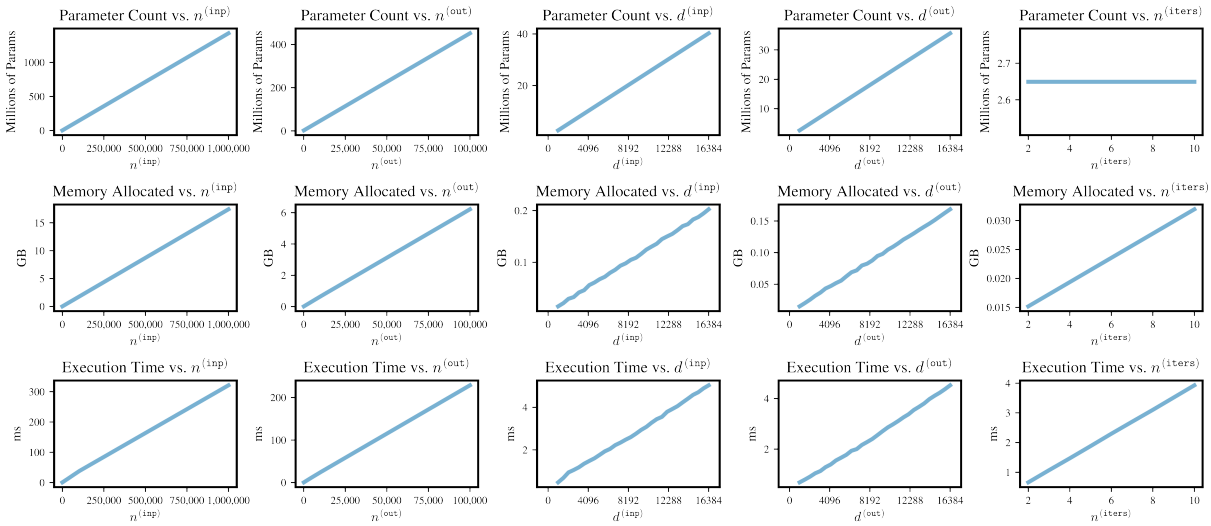

Figure 6: Parameter count, memory footprint, and execution time of a forward pass as we vary each of $n^{(\text{inp})}$, $n^{(\text{out})}$, $d^{(\text{inp})}$, $d^{(\text{out})}$, and $n^{(\text{iters})}$, while keeping the others constant at a baseline, at 32-bit precision, keeping track of gradients, on a recent hardware accelerator (GPU). Baseline values are 100, 100, 1024, 1024, and 2, respectively. Memory figures are peak allocations.

5 Experiments

5.1 Efficiency and Scalability

We measure our implementation's parameter count, memory footprint, and execution time as we increase $n^{(\text{inp})}$ from 100 to 1,000,000 input vectors, $n^{(\text{out})}$ from 100 to 100,000 output vectors, $d^{(\text{inp})}$ from 1024 to 16384 elements per input vector, $d^{(\text{out})}$ from 1024 to 16384 elements per output vector, and number of iterations $n^{(\text{iters})}$ from 2 to 10. We find that parameter count, memory footprint, and execution time are linear in each of $n^{(\text{inp})}$, $n^{(\text{out})}$, $d^{(\text{inp})}$, and $d^{(\text{out})}$ (Figure 6), enabling fine-grained control over memory consumption and computational cost. Given a memory and compute budget, we can increase the maximum length of input sequences our implementation can route by reducing output sequence length, and vice versa. Memory footprint and execution time are also linear in the number of iterations.

We also compare our implementation's parameter count, memory footprint, and execution time to those of a Transformer encoder layer using self-attention as we increase sequence length up to 2000 vectors, keeping vector size constant at 1024. To make the comparison possible, we restrict our implementation to input and output sequences that have the same shape, routing over two iterations, the fewest possible. We find our implementation requires fewer parameters for sequences with up to 600 vectors, allocates less memory for sequences with up to 800 vectors, and incurs less computation for sequences with up to 1700 vectors (Figure 7), which is surprising to us, because our algorithm proposes $n^{(\text{out})}$ output vectors per input vector, whereas the query-key-value mechanism of self-attention proposes only one output vector (a "value") per input vector.

5.2 Performance on Benchmarks

We test our implementation on six classification benchmarks in natural language and vision, obtaining accuracy that is competitive with, and in one case better than, the state of the art (Table 2). For each benchmark, we add a classification head to a pretrained Transformer. The head accepts as input all token embeddings computed by every Transformer layer, flattens them into a single sequence, and sequentially applies three routings:

	$n^{(\text{inp})}$	$n^{(\text{out})}$	$d^{(\text{inp})}$	$d^{(\text{out})}$
R_1	–	$n^{(\text{hid})}$	$d^{(\text{emb})}$	$d^{(\text{hid})}$
R_2	$n^{(\text{hid})}$	$n^{(\text{hid})}$	$d^{(\text{hid})}$	$d^{(\text{hid})}$
R_3	$n^{(\text{hid})}$	$n^{(\text{cls})}$	$d^{(\text{hid})}$	1

where R_1's number of input vectors is unspecified because the flattened sequence's length is variable, $n^{(\text{hid})}$ is a number of hidden explanatory vectors of our choosing, $d^{(\text{emb})}$ is the pretrained Transformer's embedding size, $d^{(\text{hid})}$ is the size of

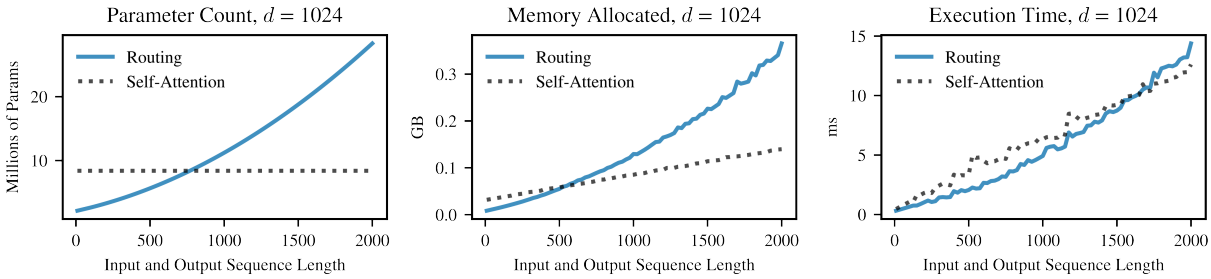

Figure 7: Comparison to a Transformer encoder layer using self-attention. To make comparison possible, we restrict our implementation to $n^{(\texttt{inp})} = n^{(\texttt{out})}$, $d^{(\texttt{inp})} = d^{(\texttt{out})}$, and $n^{(\texttt{iters})} = 2$. Data is for a forward pass on a recent hardware accelerator (GPU) at 32-bit precision, keeping track of gradients, using dense Softmax functions. Self-attention uses eight heads, the default. Memory figures are peak allocations.

the hidden explanatory vectors, and $n^{(\texttt{cls})}$ is the number of classes specific to each task.

For natural language tasks, we use RoBERTa-large (Liu et al., 2019) as the pretrained Transformer. For visual tasks, we use BEiT-large with 16×16 patches from 224×224 images (Bao et al., 2021). We freeze the Transformer. For all tasks, we specify $n^{(\texttt{hid})} = 64$ and $d^{(\texttt{hid})} = d^{(\texttt{emb})}$. All routings execute $n^{(\texttt{iters})} = 2$ iterations, the fewest possible. Before flattenning the hidden embeddings we apply layer normalization at each level of depth. If the input sequence's length is greater than the Transformer's maximum sequence length, we split the input sequence into chunks, apply the Transformer to each chunk, and join the hidden states computed for all chunks at every level of Transformer depth. The longest flattened sequence we see among all benchmarks has 89,600 input vectors, computed by RoBERTa-large's 25 hidden layers for a natural language sample drawn from the IMDB movie review dataset, split in 7 chunks, each with 512 subword tokens.

Classification Benchmark		Accuracy (%)
Natural Language		
IMDB		96.2
SST-5*		59.8
SST-2		96.0
Vision		
ImageNet-1K @ 224×224	Top1	86.7
	Top5	98.1
CIFAR-100		93.8
CIFAR-10		99.2

* New state-of-the-art accuracy.

Table 2: Classification accuracy.

5.3 End-to-End Credit Assignments

Vectors remain independent of each other between each routing executed in the classification head, so we can compute end-to-end credit assignments for all benchmark tasks. Each head executes three routings, giving us three credit assignment matrices. We multiply them as described in Appendix A, obtaining a matrix of end-to-end credit assigned to every hidden Transformer embedding i for each predicted classification score j:

$$\phi_{ij}^{(\texttt{e2e})} \longleftarrow \frac{\sum_{j'j''} \phi_{ij'}^{(\texttt{R}_1)} \phi_{j'j''}^{(\texttt{R}_2)} \phi_{j''j}^{(\texttt{R}_3)}}{\sigma\left(\sum_{j'j''} \phi_{ij'}^{(\texttt{R}_1)} \phi_{j'j''}^{(\texttt{R}_2)} \phi_{j''j}^{(\texttt{R}_3)}\right)}, \qquad (15)$$

where $j' = (1, 2, \ldots, n^{(\texttt{hid})})$ and $j'' = (1, 2, \ldots, n^{(\texttt{hid})})$, and σ computes the standard deviation over all elements, scaling the credit assignments to unit variance. The largest end-to-end credit assignment matrix we see among all benchmarks has 4925×1000 elements, consisting of the end-to-end credit assigned to embeddings of a special token and 196 image patches computed by each of BEiT-large's 25 levels of depth, in a flattened sequence with 4925 input vectors, for 1000 predicted scores, each an output vector with one element, for ImageNet-1K classification.

We sum $\phi_{ij}^{(\texttt{e2e})}$'s elements over all levels of Transformer depth to obtain the credit assigned to subword tokens and pixel patches, and over groups of tokens and patches to obtain the credit assigned to sentences and image regions. We find the end-to-end credit assignments are interpretable. Figures 8 and 9 show typical examples.

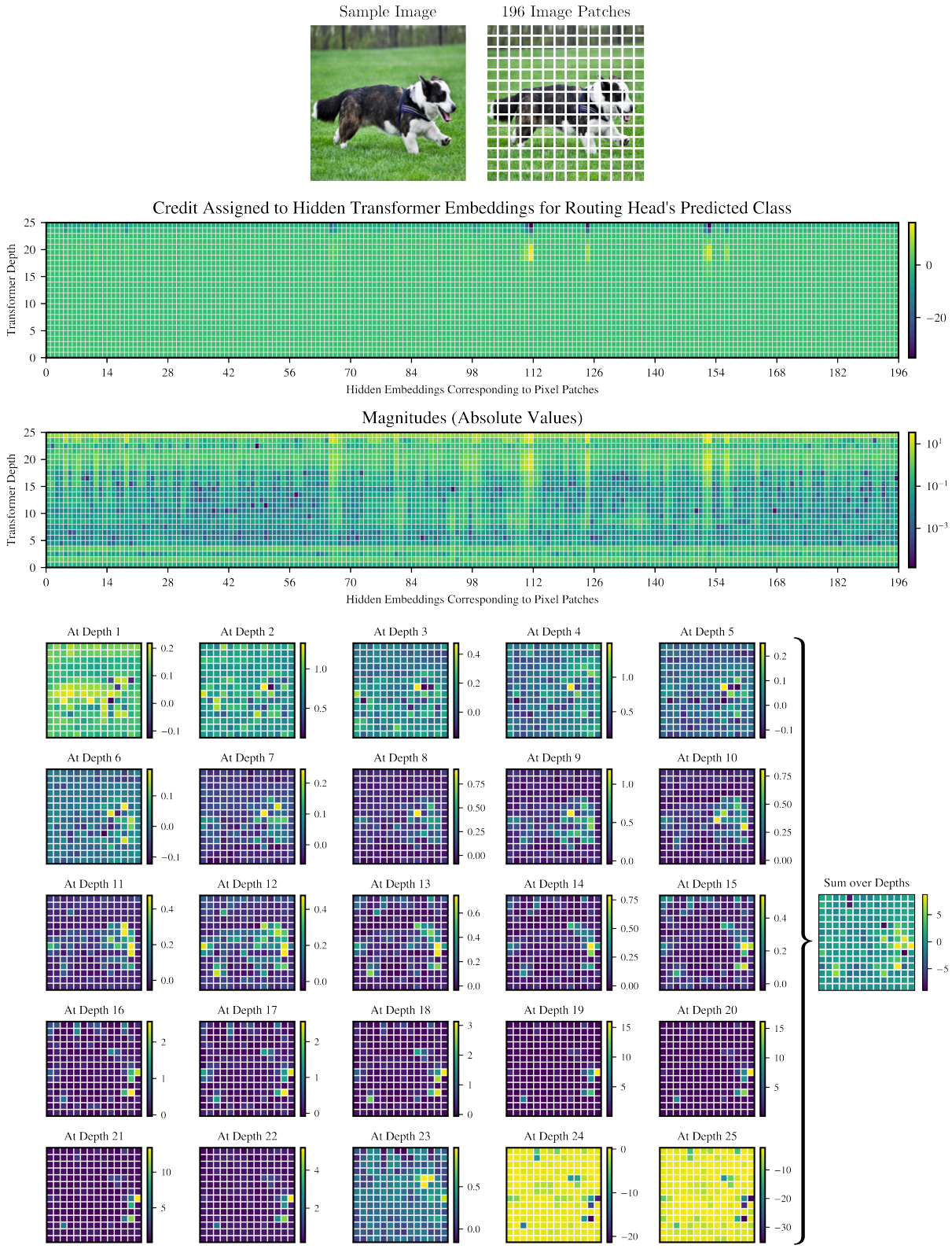

Figure 8: Typical example of end-to-end credit assigned to Transformer hidden states in a visual task. Here, our three-layer routing head assigns credit to the dog's entire body in shallower layers, and to its nose, mouth, ears, and paws in deeper layers. The matrix of end-to-end credit assignments $\phi_{ij}^{(\mathsf{e2e})}$ has 4925×1000 elements, consisting of credit assigned to 197 hidden embeddings at 25 levels of Transformer depth, or 4925 input vectors, for 1000 classification scores, each an output vector with one element. We show the absolute values of 4900 credit assignments to embeddings corresponding to 196 image patches, for the highest score, excluding 25 credit assignments to a special token added to the input sequence.

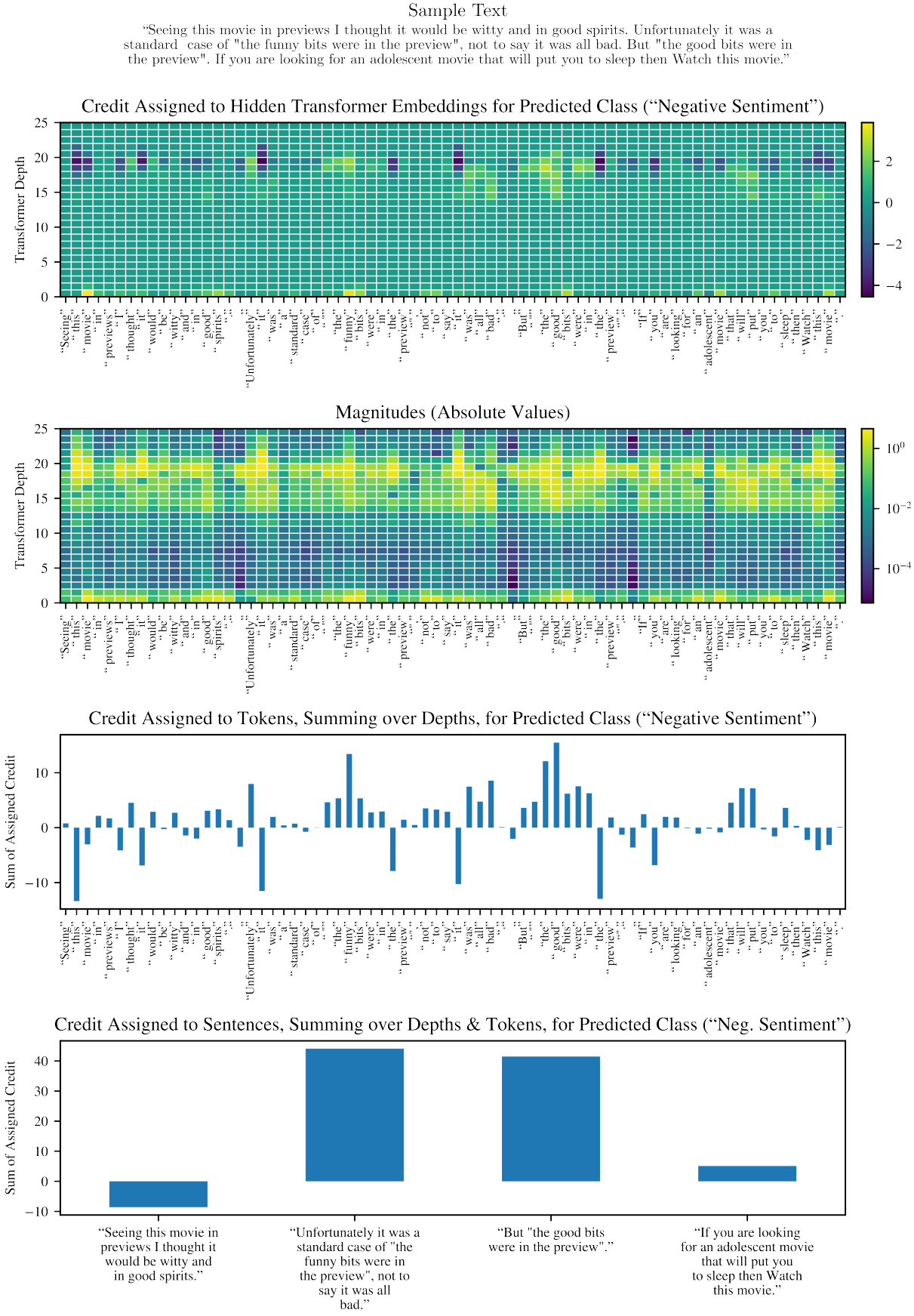

Figure 9: Typical example of end-to-end credit assigned to Transformer hidden states in a natural language task. Here, $\phi_{ij}^{(\mathsf{e2e})}$ has 1850×2 elements, consisting of credit assigned to 74 hidden embeddings at 25 levels of Transformer depth, or 1850 input vectors, for 2 classification scores, each an output vector with one element. We show 1800 credit assignments to embeddings corresponding to 72 subword tokens for the highest score, excluding 50 credit assignments to two special tokens added to the input sequence.

References

Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. 2019. Deep equilibrium models. *CoRR* abs/1909.01377.

Hangbo Bao, Li Dong, and Furu Wei. 2021. Beit: BERT pre-training of image transformers. *CoRR* abs/2106.08254.

Zi-Yi Dou, Zhaopeng Tu, Xing Wang, Longyue Wang, Shuming Shi, and Tong Zhang. 2019. Dynamic layer aggregation for neural machine translation with routing-by-agreement. *CoRR* abs/1902.05770.

Taeyoung Hahn, Myeongjang Pyeon, and Gunhee Kim. 2019. Self-routing capsule networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*. Curran Associates, Inc., volume 32.

Franz A. Heinsen. 2019. An algorithm for routing capsules in all domains. *CoRR* abs/1911.00792.

Geoffrey Hinton, Sara Sabour, and Nicholas Frosst. 2018. Matrix capsules with em routing. In *International Conference on Learning Representations (ICLR)*.

Dmitry Krotov and John Hopfield. 2021. Large associative memory problem in neurobiology and machine learning. *CoRR* abs/1710.09829.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR* abs/1907.11692.

Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, Curran Associates, Inc., pages 4765–4774.

Marvin Minsky. 1986. *The Society of Mind*. Simon and Schuster, Inc., USA.

Jathushan Rajasegaran, Vinoj Jayasundara, Sandaru Jayasekara, Hirunima Jayasekara, Suranga Seneviratne, and Ranga Rodrigo. 2019. Deepcaps: Going deeper with capsule networks. *CoRR* abs/1904.09546.

Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil Sandve, Victor Greiff, David Kreil, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. 2021. Hopfield networks is all you need. *CoRR* abs/2008.02217.

Fabio De Sousa Ribeiro, Georgios Leontidis, and Stefanos D Kollias. 2020. Capsule routing via variational bayes. In *AAAI*. pages 3749–3756.

Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. 2017. Dynamic routing between capsules. *CoRR* abs/1710.09829.

Yao-Hung Hubert Tsai, Nitish Srivastava, Hanlin Goh, and Ruslan Salakhutdinov. 2020. Capsules with inverted dot-product attention routing. In *International Conference on Learning Representations (ICLR)*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR* abs/1706.03762.

Dilin Wang and Qiang Liu. 2018. An optimization view on dynamic routing between capsules. In *International Conference on Learning Representations (ICLR)*.

Zhang Xinyi and Lihui Chen. 2019. Capsule graph neural network. In *International Conference on Learning Representations (ICLR)*.

Suofei Zhang, Wei Zhao, Xiaofu Wu, and Quan Zhou. 2018. Fast dynamic routing based on weighted kernel density estimation. *CoRR* abs/1805.10807.

A Composability of Credit Assignments

If each input vector's votes are independent of other input vectors' votes, then we can compose the "bang per bit" credit-assignment coefficients ϕ_{ij} on their own, independently of data transformations. Here, we show methods for computing end-to-end credit assignments over four common compositions of routings.[8]

A.1 In Sequential Routings

If we compose the sequential application of two routings, R_1 and R_2, into a neural network,

$$x_{jh}^{(\mathtt{out})} \longleftarrow \mathsf{R}_2(\mathsf{R}_1(x_{id}^{(\mathtt{inp})})), \tag{16}$$

we can obtain the neural network's end-to-end credit assignments $\phi_{ij}^{(\mathtt{e2e})}$, over both routings, by multiplying their credit-assignment matrices,

$$\phi_{ij}^{(\mathtt{e2e})} \longleftarrow \sum_{j'} \phi_{ij'}^{(\mathsf{R}_1)} \phi_{j'j}^{(\mathsf{R}_2)}, \tag{17}$$

where $\phi_{ij'}^{(\mathsf{R}_1)}$ and $\phi_{j'j}^{(\mathsf{R}_2)}$ are the credit-assignment matrices computed by R_1 and R_2, respectively, and j' is the common index over R_1's output vectors and R_2's input vectors.

For a longer sequence of routings, we can obtain end-to-end credit assignments by multiplying the corresponding chain of credit-assignment matrices, as matrix multiplication is associative.

A.2 In Residual Routings

If we apply one routing as a residual to another,

$$x_{jh}^{(\mathtt{out})} \longleftarrow \mathsf{R}_1(x_{id}^{(\mathtt{inp})}) + \mathsf{R}_2(\mathsf{R}_1(x_{id}^{(\mathtt{inp})})), \tag{18}$$

we can obtain end-to-end credit assignments $\phi_{ij}^{(\mathtt{e2e})}$ by adding the product of the two credit-assignment matrices to the first one,

$$\phi_{ij}^{(\mathtt{e2e})} \longleftarrow \phi_{ij}^{(\mathsf{R}_1)} + \sum_{j'} \phi_{ij'}^{(\mathsf{R}_1)} \phi_{j'j}^{(\mathsf{R}_2)}, \tag{19}$$

where $\phi_{ij}^{(\mathsf{R}_1)}$ and $\phi_{j'j}^{(\mathsf{R}_2)}$ are the credit assignment matrices computed by R_1 and R_2, respectively, and $j' = j$ (necessary for disambiguation).

For a sequence of residual routings, we can obtain end-to-end credit assignments by multiplying each additional residual credit-assignment matrix with, and then adding the result back to, the previous state of the end-to-end credit-assignment matrix, as matrix addition is associative.

[8] Subject to the same condition of independence, our methods apply also to modern Hopfield networks with bipartite structure, including Transformer self-attention, as they are simplifications of our routing algorithm. See 3.3.

A.3 In Sums of Routings

If we sum two independent routings R_1 and R_2,

$$x_{jh}^{(\mathtt{out})} \longleftarrow \mathsf{R}_1(x_{i_1 d_1}^{(\mathtt{inp1})}) + \mathsf{R}_2(x_{i_2 d_2}^{(\mathtt{inp2})}), \tag{20}$$

we can obtain end-to-end credit assignments by concatenating the two credit-assignment matrices over their mutually exclusive indices,

$$\phi_{ij}^{(\mathtt{e2e})} \longleftarrow \phi_{i_1 j}^{(\mathsf{R}_1)} \oplus \phi_{i_2 j}^{(\mathsf{R}_2)} = \begin{bmatrix} \phi_{i_1 j}^{(\mathsf{R}_1)} \\ \phi_{i_2 j}^{(\mathsf{R}_2)} \end{bmatrix}, \tag{21}$$

where $\phi_{i_1 j}^{(\mathsf{R}_1)}$ and $\phi_{i_2 j}^{(\mathsf{R}_2)}$ are the credit assignment matrices computed by R_1 and R_2, respectively, the symbol $\oplus$ denotes a direct sum over mutually exclusive indices i_1 and i_2, and $i = (i_1; i_2)$ is a single index that concatenates indices i_1 and i_2.

For sums of three or more independent routings, we can obtain end-to-end credit assignments by concatenating their credit-assignment matrices over the mutually exclusive input indices, but we must fix the order of concatenation, as direct sums are associative but not commutative.

A.4 In Concatenations of Routings

If we concatenate the output vectors of two independent routings R_1 and R_2,

$$\begin{aligned} x_{j_1 h}^{(\mathtt{hid1})} &\longleftarrow \mathsf{R}_1(x_{i_1 d_1}^{(\mathtt{inp1})}) \\ x_{j_2 h}^{(\mathtt{hid2})} &\longleftarrow \mathsf{R}_2(x_{i_2 d_2}^{(\mathtt{inp2})}) \\ x_{jh}^{(\mathtt{out})} &\longleftarrow x_{j_1 h}^{(\mathtt{hid1})} \oplus x_{j_2 h}^{(\mathtt{hid2})}, \end{aligned} \tag{22}$$

where $j = (j_1; j_2)$ is the concatenated index, we can obtain end-to-end credit assignments with a direct sum of the credit-assignment matrices,

$$\phi_{ij}^{(\mathtt{e2e})} \longleftarrow \phi_{i_1 j_1}^{(\mathsf{R}_1)} \oplus \phi_{i_2 j_2}^{(\mathsf{R}_2)} = \begin{bmatrix} \phi_{i_1 j_1}^{(\mathsf{R}_1)} & 0 \\ 0 & \phi_{i_2 j_2}^{(\mathsf{R}_2)} \end{bmatrix}, \tag{23}$$

where $\phi_{i_1 j_1}^{(\mathsf{R}_1)}$ and $\phi_{i_2 j_2}^{(\mathsf{R}_2)}$ are the credit assignment matrices computed by R_1 and R_2, respectively, $\oplus$ again denotes a direct sum over mutually exclusive indices, $i = (i_1; i_2)$, and $j = (j_1; j_2)$.

For three or more concatenations, we can obtain end-to-end credit assignments with direct sums over mutually exclusive input indices, provided we fix the order of concatenation, as direct sums are associative but not commutative.

B Derivation of Update Rule

$$
\begin{aligned}
\mathcal{U}(\,\cdot\,|x_{id}^{(\text{inp})}) &= \sum_i \beta_{ij}^{(\text{use})} D_{ij}^{(\text{use})} V_{ijh} - \sum_i \beta_{ij}^{(\text{ign})} D_{ij}^{(\text{ign})} V_{ijh} \\
&= \sum_i \left(\beta_{ij}^{(\text{use})} D_{ij}^{(\text{use})} V_{ijh} - \beta_{ij}^{(\text{ign})} D_{ij}^{(\text{ign})} V_{ijh} \right) \\
&= \sum_i \left(\beta_{ij}^{(\text{use})} \left(f(a_i^{(\text{inp})}) R_{ij} \right) V_{ijh} - \beta_{ij}^{(\text{ign})} \left(f(a_i^{(\text{inp})}) - f(a_i^{(\text{inp})}) R_{ij} \right) V_{ijh} \right) \\
&= \sum_i \left(\beta_{ij}^{(\text{use})} f(a_i^{(\text{inp})}) R_{ij} V_{ijh} - \beta_{ij}^{(\text{ign})} f(a_i^{(\text{inp})}) V_{ijh} + \beta_{ij}^{(\text{ign})} f(a_i^{(\text{inp})}) R_{ij} V_{ijh} \right) \\
&= \sum_i \left(R_{ij} (\beta_{ij}^{(\text{use})} + \beta_{ij}^{(\text{ign})}) f(a_i^{(\text{inp})}) V_{ijh} - \beta_{ij}^{(\text{ign})} f(a_i^{(\text{inp})}) V_{ijh} \right) \\
&= \sum_i \Bigg(R_{ij} \underbrace{(\beta_{ij}^{(\text{use})} + \beta_{ij}^{(\text{ign})}) f(\mathcal{A}(x_{id}^{(\text{inp})})) \mathcal{F}(x_{id}^{(\text{inp})})}_{\text{Define as } \mathcal{M}(x_{id}^{(\text{inp})})} - \underbrace{\beta_{ij}^{(\text{ign})} f(\mathcal{A}(x_{id}^{(\text{inp})})) \mathcal{F}(x_{id}^{(\text{inp})})}_{\text{Define as } \mathcal{B}(x_{id}^{(\text{inp})})} \Bigg) \\
&= \sum_i \left(\underbrace{\frac{e^{\mathcal{S}(x_{id}^{(\text{inp})}, \mathcal{G}(\cdot))}}{\sum_j e^{\mathcal{S}(x_{id}^{(\text{inp})}, \mathcal{G}(\cdot))}}}_{\text{Define as } \mathcal{R}(\,\cdot\,|x_{id}^{(\text{inp})})} \mathcal{M}(x_{id}^{(\text{inp})}) - \mathcal{B}(x_{id}^{(\text{inp})}) \right) \\
&= \sum_i \Big(\mathcal{R}(\,\cdot\,|\underbrace{x_{id}^{(\text{inp})}}_{\text{Keys}}) \underbrace{\mathcal{M}(x_{id}^{(\text{inp})})}_{\text{Values}} - \underbrace{\mathcal{B}(x_{id}^{(\text{inp})})}_{\text{Biases}} \Big)
\end{aligned}
$$

// $\mathcal{R}$ computes each iteration's *routing probabilities*.
// $\mathcal{M}$ obtains content-addressable *memory values*.
// $\mathcal{B}$ obtains content-addressable *memory biases*.

C Efficient Lazy Contraction of Votes

$$
\begin{aligned}
\sum_i \phi_{ij} V_{ijh} &= \sum_i \phi_{ij} \mathcal{F}_2\left(\mathcal{F}_1\left(x_{id}^{(\text{inp})}\right)\right) && \text{// Lazy evaluation in each iteration.} \\
&= \sum_i \phi_{ij} \Bigg(\underbrace{\sum_d W_{dh}^{(\mathcal{F}_2)} \underbrace{\left(\frac{x_{id}^{(\text{inp})} W_{jd}^{(\mathcal{F}_1)}}{\sqrt{n^{(\text{inp})}}} \right)}_{\mathcal{O}(n^{(\text{inp})} \times n^{(\text{out})} \times d^{(\text{inp})})}}_{\mathcal{O}(n^{(\text{inp})} \times n^{(\text{out})} \times d^{(\text{out})})} + B_{jh}^{(\mathcal{F}_2)} \Bigg)_{\mathcal{O}(n^{(\text{inp})} \times n^{(\text{out})} \times d^{(\text{out})})} && \begin{array}{l}\text{// If we evaluate expression naively,} \\ \text{// all intermediate tensors occupy} \\ \text{// either } \mathcal{O}(n^{(\text{inp})} n^{(\text{out})} d^{(\text{inp})}) \text{ or} \\ \text{// } \mathcal{O}(n^{(\text{inp})} n^{(\text{out})} d^{(\text{out})}) \text{ space.}\end{array} \\
&= \sum_i \phi_{ij} \sum_d W_{dh}^{(\mathcal{F}_2)} \frac{x_{id}^{(\text{inp})} W_{jd}^{(\mathcal{F}_1)}}{\sqrt{n^{(\text{inp})}}} + \sum_i \phi_{ij} B_{jh}^{(\mathcal{F}_2)} && \text{// Distribute contraction with } \phi_{ij}. \\
&= \frac{1}{\sqrt{n^{(\text{inp})}}} \sum_i \phi_{ij} \sum_d W_{dh}^{(\mathcal{F}_2)} x_{id}^{(\text{inp})} W_{jd}^{(\mathcal{F}_1)} + \sum_i \phi_{ij} B_{jh}^{(\mathcal{F}_2)} && \text{// Factor out scalar in first term.} \\
&= \frac{1}{\sqrt{n^{(\text{inp})}}} \sum_{id} W_{dh}^{(\mathcal{F}_2)} W_{jd}^{(\mathcal{F}_1)} \phi_{ij} x_{id}^{(\text{inp})} + \sum_i \phi_{ij} B_{jh}^{(\mathcal{F}_2)} && \begin{array}{l}\text{// Express first term as a sequence} \\ \text{// of elementwise tensor operations} \\ \text{// contracted over two indices, } id.\end{array} \\
&= \frac{1}{\sqrt{n^{(\text{inp})}}} \sum_d W_{dh}^{(\mathcal{F}_2)} W_{jd}^{(\mathcal{F}_1)} \sum_i \phi_{ij} x_{id}^{(\text{inp})} + \sum_i \phi_{ij} B_{jh}^{(\mathcal{F}_2)} && \begin{array}{l}\text{// Contract over } i \text{ before multiplying} \\ \text{// elementwise by } W_{jd}^{(\mathcal{F}_1)}.\end{array} \\
&= \frac{1}{\sqrt{n^{(\text{inp})}}} \Bigg(\underbrace{\sum_d W_{dh}^{(\mathcal{F}_2)} \underbrace{\left(W_{jd}^{(\mathcal{F}_1)} \underbrace{\left(\sum_i \phi_{ij} x_{id}^{(\text{inp})} \right)}_{\mathcal{O}(n^{(\text{out})} \times d^{(\text{inp})})} \right)}_{\mathcal{O}(n^{(\text{out})} \times d^{(\text{inp})})}}_{\mathcal{O}(n^{(\text{out})} \times d^{(\text{out})})} + \underbrace{\sum_i \phi_{ij} B_{jh}^{(\mathcal{F}_2)}}_{\mathcal{O}(n^{(\text{out})} \times d^{(\text{out})})} \Bigg) && \begin{array}{l}\text{// Now, all intermediate tensors} \\ \text{// occupy either } \mathcal{O}(n^{(\text{out})} d^{(\text{inp})}) \\ \text{// or } \mathcal{O}(n^{(\text{out})} d^{(\text{out})}) \text{ space, and} \\ \text{// we apply } \mathcal{F}_2 \text{ as a last step} \\ \text{// only once per output vector.}\end{array}
\end{aligned}
$$