Neural Circuit Diagrams: Standardized Diagrams for Deep Learning Architectures

Anonymous authors
Paper under double-blind review

Abstract

Diagrams matter. For deep learning, unfortunately, there is no standard method for blueprinting architectures. This makes deep learning architectures more difficult to understand, implement, and analyse than they should be. In this paper, we introduce *neural circuit diagrams*, which reliably visualize the inner workings of deep learning architectures. Neural circuit diagrams are introduced in an accessible manner, meaning designers can use them and point their readers to this work without placing an undue burden. Neural circuit diagrams are presented with figures of gradually increasing complexity, building up to a full diagram of the transformer architecture from *Attention is All You Need.* This lets us show how neural circuit diagrams can be used in practice and highlights their advantage over standard methods of communicating architectures.

1 Introduction

Deep learning models are statistical engines that rely on various components connected in complicated ways. Data passes through a network and is altered, rearranged, and transformed. Training slowly nudges these alterations until inputs imitate desired outputs. Currently, these machines' mechanisms are expressed by a combination of linear algebra (Goodfellow et al., 2016; Chiang et al., 2021) and makeshift diagrams (Vaswani et al., 2017) created for specific situations. This is sufficient to propose ideas, even with some technical detail, but is insufficient for the reliable implementation and analysis we should aim for. This paper is not the first to make this criticism. The work on named tensors (Chiang et al., 2021) and formal descriptions of transformers (Phuong & Hutter, 2022) identifies this same problem: that the deep learning community suffers from ambiguously presented architectures. Our solution are neural circuit diagrams, which allow architectures to be unambiguously visualised, offering a major improvement over current means of communicating architectures.

1.1 The Need for Better Diagrams

This paper aims to justify the necessity of neural circuit diagrams and to present their proper usage in an accessible manner. To this end, we will pay special attention to clearly diagramming the architecture from the paper *Attention is All You Need* (Vaswani et al., 2017). It is the keystone paper that introduced transformer architectures, which have revolutionized language and image processing (Devlin et al., 2018; Dosovitskiy et al., 2021; Lin et al., 2021). However, the original paper's often ambiguous expression has been previously noted and attempted to be addressed (Chiang et al., 2021; Phuong & Hutter, 2022).

Its ambiguity is still relevant; more recent work has continued relying on its unclear equations and diagrams (Lin et al., 2021). Therefore, we know there is a need to more clearly present the ideas in *Attention is All You Need.* In Section 4.1, the shortfalls of the equations and diagrams in *Attention is All You Need* are outlined, and in Section 4.2, we present our alternative approach. By contrasting our diagrams with those from the original paper, the benefits of neural circuit diagrams become evident.

Furthermore, by building up all the tools needed to understand the architecture of *Attention is All You Need*, the reader will have acquired a thorough understanding of the original transformer architecture and the techniques needed to understand other architectures. The original transformer architecture has many key components connected in sophisticated ways: fully connected layers, copying, repeated layers, residual networks (He et al., 2015), layer normalization (Ba et al., 2016), among others. We will learn how these components are represented in neural circuit diagrams, clarifying complex relationships. This forms the foundation for a robust understanding of other architectures which use similar components and have the same need for clear, visual presentation.

The awareness of the shortfalls in *Attention is All You Need* is a byproduct of the scrutiny of fame rather than its unique problems. The tools it uses for its explanations, linear algebra notation for equations and makeshift diagrams to show the relationship between components, are standard in the deep learning field and can be found in other papers' descriptions of architectures (Rombach et al., 2022). Linear algebra is excellent in the specialised domain of lots of data organised along two axes interacting over linear operations. This domain covers many important problems and allows for minimal and intuitive notation. As a result, linear algebra notation is widely known, tempting its use. However, deep learning extensively uses non-linear operations and arranges data in various ways that quickly exhaust the limitations of linear algebra, which is clear in the ambiguous equations of *Attention is All You Need.* The use of linear algebra notation is standard practice in deep learning (Goodfellow et al., 2016). The makeshift diagrams in *Attention is All You Need* are also standard; as no standard diagrammatic notation exists, authors and designers resort to original diagrams that can easily miss key details.

1.2 Improved Communication by Being *Visual* and *Explicit*

Many deep learning papers are difficult to understand, beyond the inherent technical challenge of the topic, due to a deficit of effective communication. This naturally arises partly because of the *curse of knowledge* (Pinker, 2014; Hayes & Bajzek, 2008; Ross et al., 1977); those who know something have difficulty identifying with someone who does not. This is especially true for technical topics (Hayes & Bajzek, 2008).

Authors, in their familiarity, can forget to include essential details, leaving readers confused. However, once the reader overcomes this confusion, they no longer empathise with their initial state and may present a similarly confusing explanation to a future learner. Neural circuit diagrams are *explicit*, just like code. They require all the critical information to be shown. Otherwise, the diagram fails even to be well-structured. The author is reminded to include all the necessary information as the diagram shows all components, their inputs, outputs, and all data transformations.

Making operations explicit has been addressed by named tensors (Chiang et al., 2021) and by a formal treatment of transformers (Phuong & Hutter, 2022). These methods are excellent for an unambiguous presentation of the operation of specific components. However, these methods' reliance on linearly assembled symbols fails to scale when it comes to representing whole architectures. Representing and understanding whole architectures is the focus of high-level architecture design, implementation, and analysis. For this use case, a visual system scales far better, and this is where neural circuit diagrams excel.

Additionally, visualisation is essential to human comprehension (Pinker, 2014; Sadoski, 1993; Borkin et al., 2015). Diagrams for deep learning architectures should ideally act similarly to blueprints for physical machines, enhancing our ability to comprehend, recall, and analyse their inner workings.

Taken together, a visual and explicit notation overcomes many difficulties in communicating deep learning models. Authors can explain their novel designs by describing what they see in diagrams, confident that all the key information will be present and that their explanations leverage their readers' visual comprehension ability. For readers, the key information is made clear, the relationships between components are unambiguous, and implementation or analysis can be readily done.

Building on the goal of effective visual communication, we will take cues from research into effective diagram design (Borkin et al., 2015). Our diagrams will use text extensively, be memorable at a glance, rely on pictograms, and be unafraid of redundancy. Instead of text, pictograms will often be used to identify components, injecting an additional visual element into diagrams. Bold colors and text boxes with no impact on the function of the diagram will be used to create a form that draws in the reader and highlights important details.

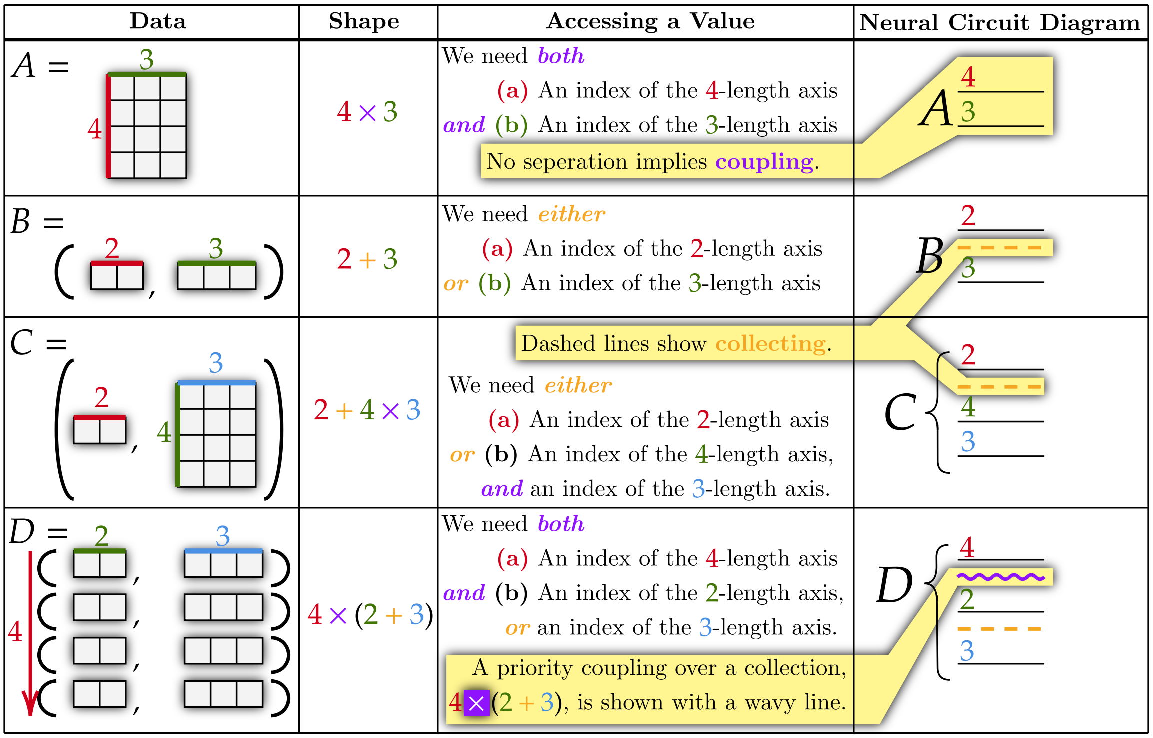
1.3 Inspiration and Related Work

The typical approach to graphically modelling data-heavy systems is with the tools of linear algebra (Abramsky & Coecke, 2008; Bridgeman & Chubb, 2017). Neural circuit diagrams are inspired by Penrose diagrams (Penrose, 1971) and monoidal string diagrams (Shiebler et al., 2021; Xu & Maruyama, 2021), which similarly represent the axes of data as separate lines. However, this work diverges from past approaches by not requiring linearity. This is achieved by having two ways of joining operations; *collecting*, a type of Cartesian product, and *coupling*, a restricted tensor product. Thus, we can create a diagrammatic system inspired by previous methods that is appropriate to the problem domain of representing deep learning architectures. While the body of this work focuses on accessibly presenting this novel approach, the appendix details how collecting and coupling can be technically defined.

2 Shaped Data and Morphisms

The fundamental data type in neural circuit diagrams is **shaped data** (see Fig. 1). Shaped data stores number values along axes of integer length. Each axis has a number of **indexes** equal to its length, which access data. Axes are combined by **coupling**, "×", or **collecting**, "+". Coupling requires an index from **both** axes to access data. For example, a table is 4 × 3-shaped data. Collecting represents data collected into independent segments and therefore needs an index from **either** axis.

Shapes can be treated as axes and further combined into higher-order shapes that are broken down in steps (see Fig. 1, C and D). Neural circuit diagrams represent shapes by a column of labelled horizontal lines, each signifying an axis. No separation between lines indicates coupling, while separation with a dashed line indicates different segments of a collection. We use letters such as i, j, and k to refer to shapes and lengths generically.



We need ***both***
(a) An index of the 4-length axis
and **(b)** An index of the 3-length axis
No seperation implies **coupling**.

We need ***either***
(a) An index of the 2-length axis
or **(b)** An index of the 3-length axis

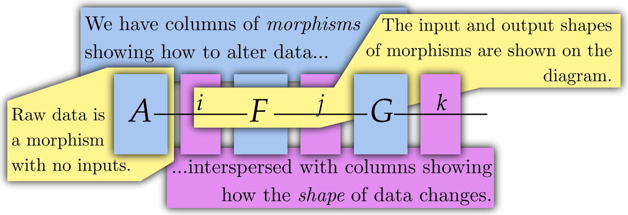
Dashed lines show **collecting**.

We need ***either***
(a) An index of the 2-length axis
or **(b)** An index of the 4-length axis,
and an index of the 3-length axis.

We need ***both***
(a) An index of the 4-length axis
and **(b)** An index of the 2-length axis,
or an index of the 3-length axis.
A priority coupling over a collection, $4 \boxed{\times} (2+3)$, is shown with a wavy line.

Figure 1: Shaped data with information about their shape, how to access values in them, and how they are presented in neural circuit diagrams.

Neural circuit diagrams consist of columns of lines representing the data shape, interspersed with columns of morphisms dictating how to modify data. **Morphisms** are operations with set input and output shapes. We illustrate them with their input shape to the left and their output shape to the right (see Fig. 2). We depict raw data as morphisms with no input to the left but an output shape to the right.



We have columns of *morphisms* showing how to alter data...

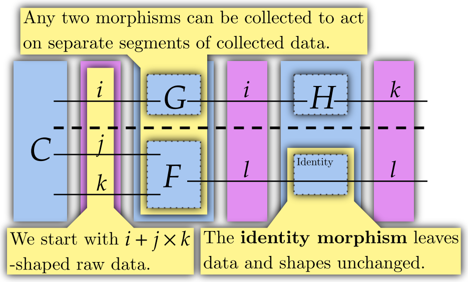
The input and output shapes of morphisms are shown on the diagram.

Raw data is a morphism with no inputs.

...interspersed with columns showing how the *shape* of data changes.

Figure 2: A basic neural circuit diagram showing raw data A of shape i fed through two morphisms F and G resulting in data of shape k.

By vertically stacking morphisms, we have them act concurrently, merging their respective inputs and outputs. We **collect** morphisms together to act separately on segments of collected data, diagrammed with a dashed line separating them (see Fig. 3). For every shape, there exists an **identity morphism** that leaves values and shapes unchanged. We illustrate the identity as a line passing through a morphism column.



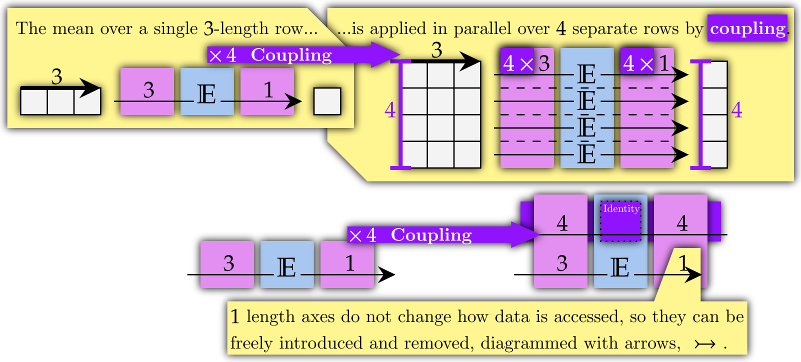
Any two morphisms can be collected to act on separate segments of collected data.

We start with $i + j \times k$ -shaped raw data.

The **identity morphism** leaves data and shapes unchanged.

Figure 3: Morphisms can be **collected** into stacks.

Coupled morphisms (see Fig. 4) operate over specific axes and in parallel with the others. This reflects the idea of applying operations over specific axes. For example, to get the mean of each row in a 4×3-table we can couple the mean operation, $\mathbb{E}$, over the 4-length row axis. This gives us a new operation, which when applied returns the mean of each row. We diagram this by extending the 4-length axis line through the morphism column, representing coupling with the identity. We can couple any morphism with the identity of any shape, in which case the identity represents leaving that shape unchanged. However, we cannot couple any two morphisms together, as the order of parallel operations may affect the result.



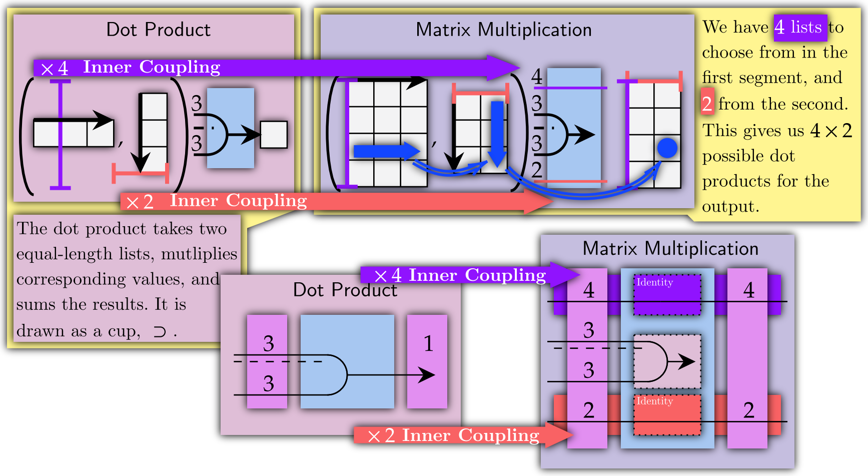
The mean over a single 3-length row...

...is applied in parallel over 4 separate rows by **coupling**

1 length axes do not change how data is accessed, so they can be freely introduced and removed, diagrammed with arrows, $\rightarrowtail$.

Figure 4: Morphisms can be **coupled** to operate in parallel with other axes.

In the case of collected inputs, such as the $4 \times 3 + 3 \times 2$-shaped data in Fig. 5, **inner coupling** lets us couple within input segments. The input segments offer many different ways that a lower-order operation can be applied. In the diagram, we extend a dot product which typically acts on $3 + 3$-shaped data to act on the collected tables. There are 4 different 3-length rows in the first segment, and 2 different 3-length columns in the second, letting us generate 4×2 separate dot products using inner coupling. Inner coupled dot products are how we represent **matrix multiplication**.

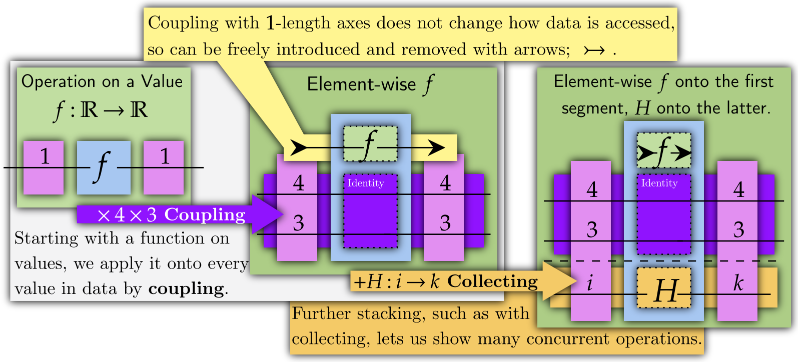


We have 4 lists to choose from in the first segment, and 2 from the second. This gives us 4×2 possible dot products for the output.

The dot product takes two equal-length lists, mutliplies corresponding values, and sums the results. It is drawn as a cup, ⊃ .

Figure 5: All possible dot products from two segments can be generated by **inner coupling**, giving **matrix multiplication**.

Operations on individual values can be coupled to apply onto every value in a segment of data, giving **element-wise operations** (see Fig. 6). Additionally, we often use **utility morphisms** (see Fig. 7) to rearrange data without altering values.


Operation on a Value
$f : \mathbb{R} \to \mathbb{R}$
1
f
1
×4×3 Coupling
Element-wise f
f
Identity
4
3
4
3
Element-wise f onto the first segment, H onto the latter.
f
Identity
4
3
4
3
i
H
k
$+H : i \to k$ Collecting

Coupling with 1-length axes does not change how data is accessed, so can be freely introduced and removed with arrows; ↣ .

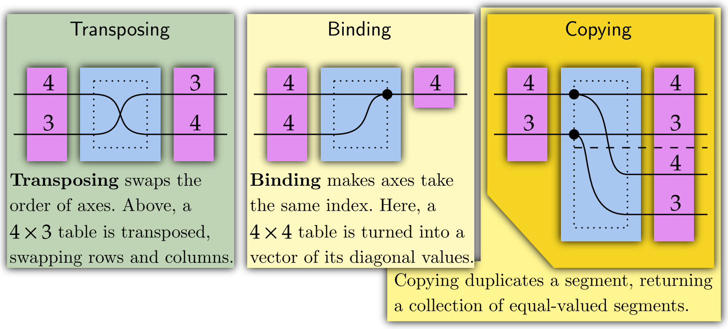
Starting with a function on values, we apply it onto every value in data by **coupling**.

Further stacking, such as with collecting, lets us show many concurrent operations.

Figure 6: A function on values is coupled to operate element-wise on data. Element-wise operations occur within segments, so in the third green rectangle f does not impact the other segment.

3 Neural Circuit Diagrams in Practice: A Basic Multi-Layer Perceptron

To understand how neural circuit diagrams can be used in practice, let's diagram a basic multi-layer perceptron. The multi-layer perceptron is a basic neural network that chains multiple **fully connected layers**. Fully connected layers are traditional neural layers. They take a vector input and produce a vector output, each element of which is a weighted sum over the input vector. These weights are learned, allowing the model to adapt to data during training. We show fully connected layers with boldface **L**. Their input and



Transposing swaps the order of axes. Above, a 4×3 table is transposed, swapping rows and columns.

Binding makes axes take the same index. Here, a 4×4 table is turned into a vector of its diagonal values.

Copying duplicates a segment, returning a collection of equal-valued segments.

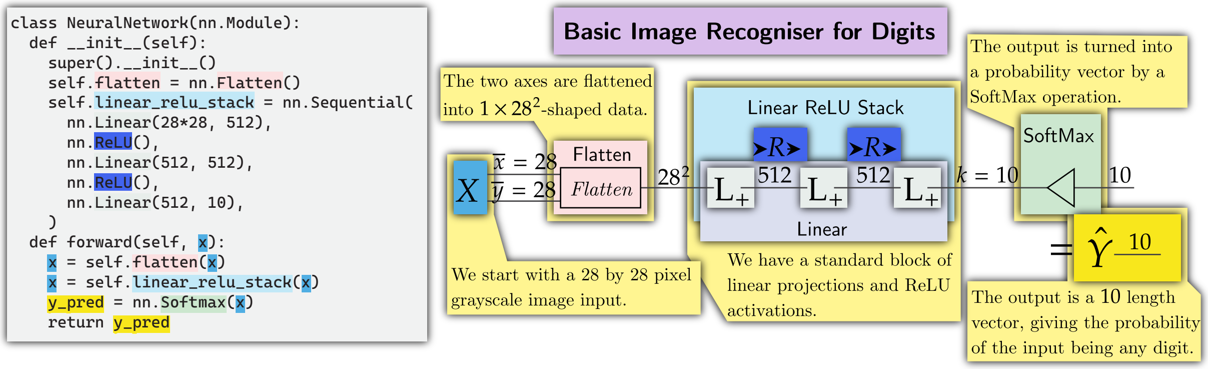
Figure 7: Some useful morphisms that rearrange data without altering values: **transposing**, **binding**, and **copying**.

output shapes are read from the diagram, and each has its own weights. They can be labelled in the top right, and adding a learned bias vector to the output is shown by a "+" in the bottom right.

Intermediate values are often passed through an *activation function*, an element-wise function (see Fig. 6) that lets intermediate values have variable sensitivities to their input data. ReLU (**re**ctified **l**inear **u**nit) activations are typically used in practice (Krizhevsky et al., 2012). It is worth noting that more recent activation functions, such as GeLU (Hendrycks & Gimpel, 2016), may have superior performance. These small architecture design details are easily shown in neural circuit diagrams. It is worth remembering that state-of-the-art designs are constantly changing. Neural circuit diagrams provide a platform to see which novel improvements can be added to old architectures.

The final output of a model often needs to be converted into probabilities. For this, a SoftMax operation is used. The operation takes a list of values and proportions them according to their exponent. This reflects important probabilistic properties. However, this operation acts over a list of values. Over a table, we need to know whether each column or each row is being proportioned. This ambiguity is unresolved by typical notation and is present in *Attention is All You Need*. Our coupling approach overcomes this issue. In diagrams, we represent SoftMax by a left-pointing triangle pictogram, ◁, representing its proportioning property. Using pictograms helps leverage human comprehension abilities (Borkin et al., 2015), and contributes to diagrams having an appealing, memorable form.

With this all established, we can present a basic image recogniser for numbers. This is taken from an introductory PyTorch tutorial, letting us see how closely related neural circuit diagrams are to PyTorch implementations.



```
class NeuralNetwork(nn.Module):
  def __init__(self):
    super().__init__()
    self.flatten = nn.Flatten()
    self.linear_relu_stack = nn.Sequential(
      nn.Linear(28*28, 512),
      nn.ReLU(),
      nn.Linear(512, 512),
      nn.ReLU(),
      nn.Linear(512, 10),
    )
  def forward(self, x):
    x = self.flatten(x)
    x = self.linear_relu_stack(x)
    y_pred = nn.Softmax(x)
    return y_pred
```

The two axes are flattened into 1×28^2-shaped data.

We start with a 28 by 28 pixel grayscale image input.

We have a standard block of linear projections and ReLU activations.

The output is turned into a probability vector by a SoftMax operation.

The output is a 10 length vector, giving the probability of the input being any digit.

Figure 8: PyTorch code and a neural circuit diagram for a basic MNIST (digit recognition) neural network taken from an introductory PyTorch tutorial. Note its close correspondence with PyTorch code.

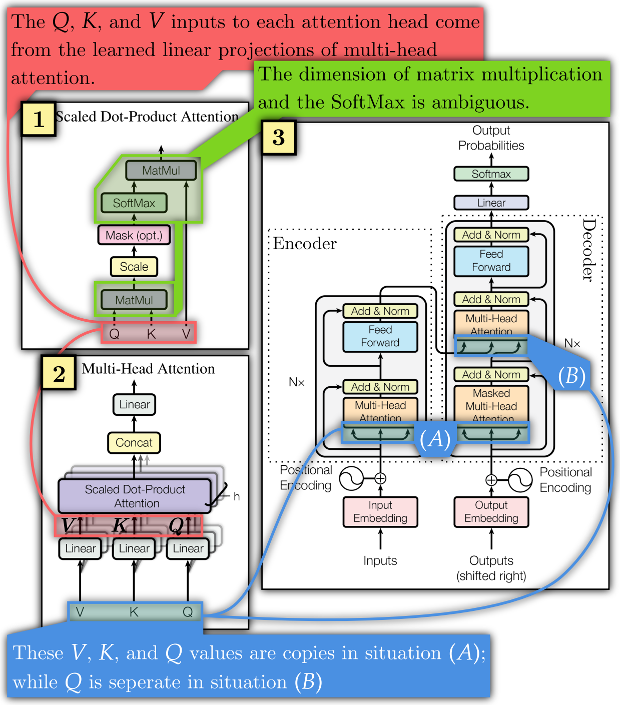
4 Attention

4.1 Shortfalls of *Attention is All You Need*

Since introduced in 2017, transformer models have revolutionized machine learning, finding applications in natural language processing, image processing, and generative tasks (Vaswani et al., 2017; Phuong & Hutter, 2022; Lin et al., 2021). Transformers' effectiveness stems in part from their ability to inject external data of arbitrary width into base data. We refer to axes representing the number of items in data as a **width**, and axes indicating information per item as a **depth**. We notate widths with an overline.

An **attention head** gives a weighted sum of the injected data's value vectors, V. The weights depend on the attention score the base data's query vectors, Q, assign to each key vector, K, of the injected data. Fully connected layers, consisting of learned matrix multiplication, generate Q, K, and V vectors from the original base and injected data. **Multi-head attention** uses multiple attention heads in parallel, enabling efficient parallel operations and the simultaneous learning of distinct attributes.

Attention is All You Need, which we refer to as the original transformer paper, explains these algorithms using diagrams (see Fig. 9) and equations (see Eqn. 1,2,3) that hinder understandability (Chiang et al., 2021; Phuong & Hutter, 2022).


The Q, K, and V inputs to each attention head come from the learned linear projections of multi-head attention.

The dimension of matrix multiplication and the SoftMax is ambiguous.

These V, K, and Q values are copies in situation (A); while Q is seperate in situation (B)

Figure 9: The diagrams from the original transformer model with my annotations. Critical information is missing regarding the origin of Q, K, and V values (**red** and **blue**) along with the axes over which operations act (**green**). Annotations in neural circuit diagrams clarify the information already present, while annotations in makeshift diagrams often introduce information not present in the diagram.

$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \ \ (d_k \text{ is the key depth}) \tag{1}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ \ldots, \text{head}_h)W^O \tag{2}$$

$$\text{where head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \tag{3}$$

The original transformer paper obscures dimension sizes and their interactions. The dimensions over which SoftMax[1] and matrix multiplication operates is ambiguous (Fig. 9.1, green; Eqn. 1, 2, 3). Determining the initial and final matrix dimensions is left to the reader. This obscures key facts required to understand transformers. For instance, K and V can have a different width to Q, allowing them to inject external information of arbitrary width. This fact is not made clear in the original diagrams or equations, yet it is necessary to understand why transformers are so effective at tasks with variable input widths, such as language processing.

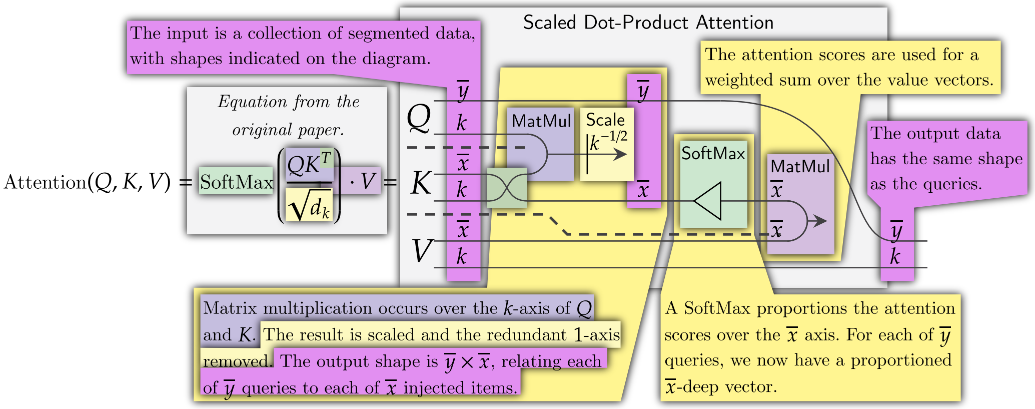
The original transformer paper also has uncertainty regarding Q, K, and V. In Fig 9.1 and Eqn. 1, they represent separate values fed to each attention head. In Fig 9.2 and Eqn. 2 and 3, they are all copies of each other at location ***(A)*** of the overall model in Fig 9.3, while Q is separate in situation ***(B)***.

Annotating makeshift diagrams does not resolve the issue of low interpretability. As they are constructed for a specific purpose by their author, they carry the author's curse of knowledge (Pinker, 2014; Hayes & Bajzek, 2008; Ross et al., 1977). As with Fig. 9, low interpretability arises from missing key information, not from insufficiently annotating the information present. Neural circuit diagrams contain all the necessary information, pointing the author to what the audience needs explained.

Though experts poring over the original transformer paper have gone on to change the world (Devlin et al., 2018; Brown et al., 2020), we aim to address its explainability shortcomings by adopting a human-centered approach focused on visual and explicit explanations. Taking full advantage of annotating the critical information already present in neural circuit diagrams, we present alternative diagrams in Fig. 10, 11, and 12.

4.2 Neural Circuit Diagrams for the Transformer Architecture

We represent specific components with pictograms. Pictograms, such as a left-pointing triangle, ◁, for the SoftMax and a vertical line, |, for scaling by $k^{-1/2}$, leverage human visual comprehension abilities (Borkin et al., 2015). The exact algorithms components employ are not the focus of neural circuit diagrams and are better left to other notational systems (Phuong & Hutter, 2022; Chiang et al., 2021). We now have the tools to diagram the attention mechanism from Eqn. 1 and Fig. 9.1 in Fig. 10.


Scaled Dot-Product Attention
Equation from the original paper.
$\text{Attention}(Q,K,V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)\cdot V =$
Q
K
V
$\overline{y}$
k
$\overline{x}$
MatMul
Scale
$k^{-1/2}$
SoftMax
MatMul

The input is a collection of segmented data, with shapes indicated on the diagram.

The attention scores are used for a weighted sum over the value vectors.

The output data has the same shape as the queries.

Matrix multiplication occurs over the k-axis of Q and K. The result is scaled and the redundant 1-axis removed. The output shape is $\overline{y} \times \overline{x}$, relating each of $\overline{y}$ queries to each of $\overline{x}$ injected items.

A SoftMax proportions the attention scores over the $\overline{x}$ axis. For each of $\overline{y}$ queries, we now have a proportioned $\overline{x}$-deep vector.

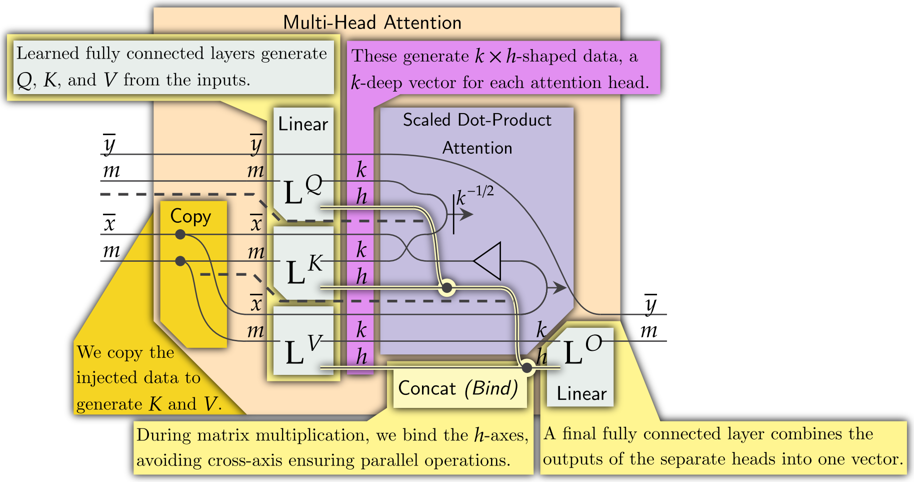
Figure 10: The original equation for attention against our diagram. The descriptions are unnecessary but clarify what is happening. Corresponds to Eqn. 1 and Fig. 9.1.

For diagramming multi-head attention in Fig. 11, we show fully connected layers with boldface **L**. Their input and output shapes are read from the diagram, and each has its own independent weights. They can be labeled in the top right, and adding a learned bias vector is shown by a "+" in the bottom right.

We conclude with the full transformer model, Fig. 12. Lines show how data evolves, while symbols and pictograms show how it is manipulated. Knowing the meaning of each symbol would be sufficient to imple-

---

[1] Using lower i and k to index over data, we have $\text{SoftMax}(\mathbf{v})_i = \exp(\mathbf{v}_i)/\Sigma_k \exp(\mathbf{v}_k)$.



Learned fully connected layers generate Q, K, and V from the inputs.

These generate $k \times h$-shaped data, a k-deep vector for each attention head.

We copy the injected data to generate K and V.

During matrix multiplication, we bind the h-axes, avoiding cross-axis ensuring parallel operations.

A final fully connected layer combines the outputs of the separate heads into one vector.

Figure 11: Neural circuit diagram for **multi-head attention**. In PyTorch, the implementation of matrix multiplication and dot products are clear with the torch.einsum function. Corresponds to Eqn. 2 and 3 and Fig. 9.2.

ment the architecture from the diagram. We use annotations to introduce new components and to provide context for what is happening.

5 Conclusion

In this work, we presented neural circuit diagrams as a human-centered approach to explaining deep learning architectures. Motivated by the insufficient detail in even groundbreaking papers, we aimed to bridge this explainability gap in a way that leverages human visual comprehension and mitigates the curse of knowledge.

We presented neural circuit diagrams with minimal mathematics, relying instead on colorful and descriptive diagrams that build up to the original transformer architecture. This approach increases accessibility, making deep learning architectures interpretable to diverse stakeholders. Additionally, designers can use neural circuit diagrams without placing an undue burden on their audience.

Future work could involve diagramming other crucial architectures, such as convolutional image processing and diffusion generative models. Examining neural circuit diagrams from a category theory perspective could help reveal the fundamental causal structure of machine learning models and allow the information flows present to be readily analysed (Shiebler et al., 2021; Perrone, 2022).

Additionally, neural circuit diagrams could clarify data models in other fields where transparency and accountability are essential, such as healthcare or finance. This route would allow for inter-disciplinary cooperation and may reveal how deep learning methods can be safely integrated into existing systems.

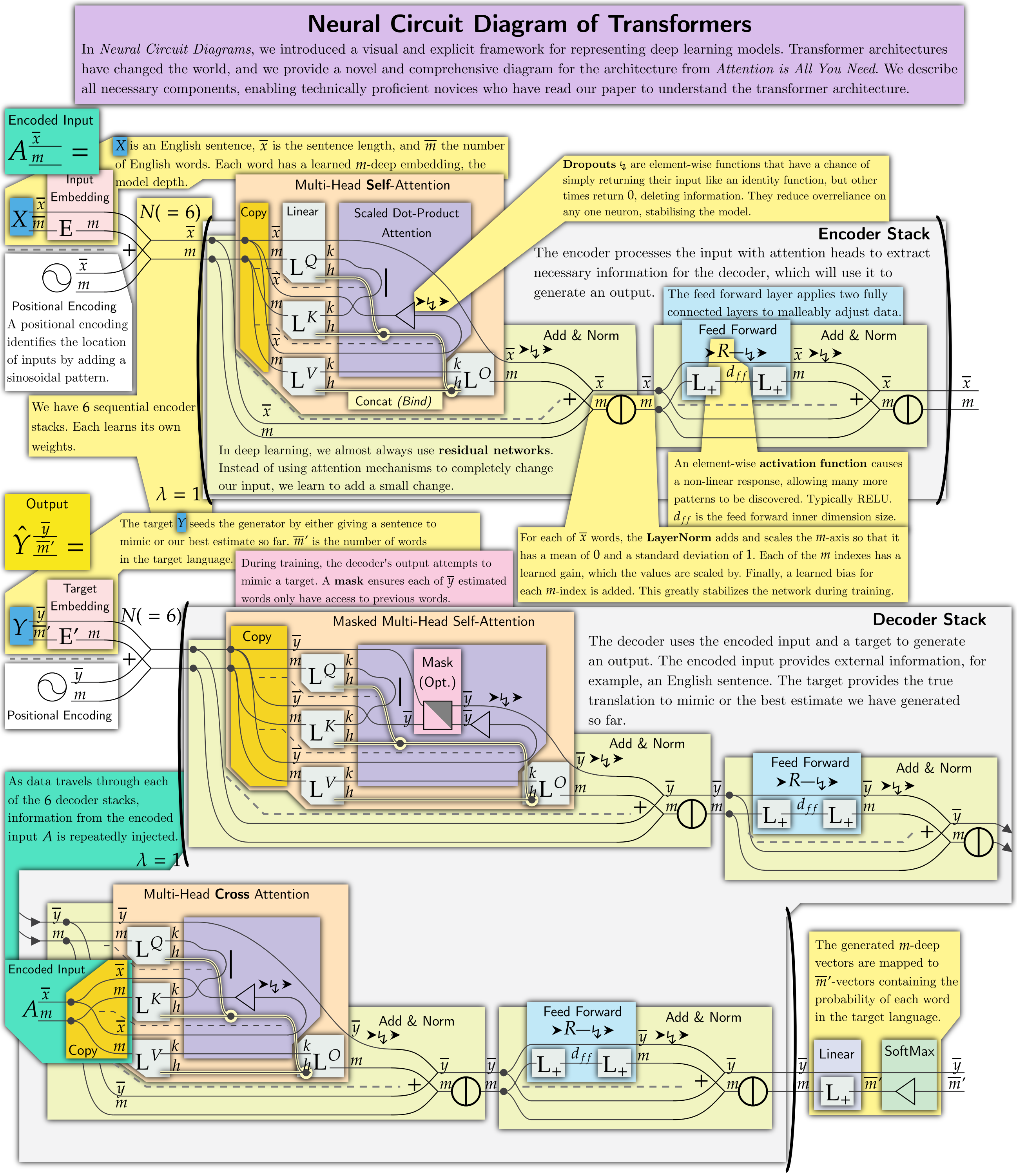
5.0.1 Acknowledgements

Mathcha was used to write equations and draw diagrams. The Harvard NLP annotated transformer was invaluable for drawing Fig. 12.

References

Samson Abramsky and Bob Coecke. *Categorical quantum mechanics.* arXiv, 2008.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.



Neural Circuit Diagram of Transformers

In *Neural Circuit Diagrams*, we introduced a visual and explicit framework for representing deep learning models. Transformer architectures have changed the world, and we provide a novel and comprehensive diagram for the architecture from *Attention is All You Need*. We describe all necessary components, enabling technically proficient novices who have read our paper to understand the transformer architecture.

X is an English sentence, $\overline{x}$ is the sentence length, and $\overline{m}$ the number of English words. Each word has a learned m-deep embedding, the model depth.

Dropouts ↯ are element-wise functions that have a chance of simply returning their input like an identity function, but other times return 0, deleting information. They reduce overreliance on any one neuron, stabilising the model.

Encoder Stack

The encoder processes the input with attention heads to extract necessary information for the decoder, which will use it to generate an output.

The feed forward layer applies two fully connected layers to malleably adjust data.

Positional Encoding

A positional encoding identifies the location of inputs by adding a sinosoidal pattern.

We have 6 sequential encoder stacks. Each learns its own weights.

In deep learning, we almost always use **residual networks**. Instead of using attention mechanisms to completely change our input, we learn to add a small change.

An element-wise **activation function** causes a non-linear response, allowing many more patterns to be discovered. Typically RELU. d_{ff} is the feed forward inner dimension size.

The target Y seeds the generator by either giving a sentence to mimic or our best estimate so far. $\overline{m}'$ is the number of words in the target language.

During training, the decoder's output attempts to mimic a target. A **mask** ensures each of $\overline{y}$ estimated words only have access to previous words.

For each of $\overline{x}$ words, the **LayerNorm** adds and scales the m-axis so that it has a mean of 0 and a standard deviation of 1. Each of the m indexes has a learned gain, which the values are scaled by. Finally, a learned bias for each m-index is added. This greatly stabilizes the network during training.

Decoder Stack

The decoder uses the encoded input and a target to generate an output. The encoded input provides external information, for example, an English sentence. The target provides the true translation to mimic or the best estimate we have generated so far.

As data travels through each of the 6 decoder stacks, information from the encoded input A is repeatedly injected.

The generated m-deep vectors are mapped to $\overline{m}'$-vectors containing the probability of each word in the target language.

Encoded Input
$A^{\overline{x}}_{\overline{m}} =$
Input Embedding
E
$N(=6)$
Multi-Head Self-Attention
Copy
Linear
Scaled Dot-Product Attention
Concat (Bind)
Add & Norm
Feed Forward
d_{ff}
$\lambda = 1$
Output
$\hat{Y}^{\overline{y}}_{\overline{m}'} =$
Target Embedding
E′
Positional Encoding
Masked Multi-Head Self-Attention
Mask (Opt.)
Multi-Head Cross Attention
Linear
SoftMax

Figure 12: The fully diagrammed architecture from *Attention is All You Need* (Vaswani et al., 2017).

M. Borkin, Z. Bylinskii, N. Kim, C. Bainbridge, C. Yeh, D. Borkin, H. Pfister, and A. Oliva. Beyond Memorability: Visualization Recognition and Recall. *IEEE Transactions on Visualization and Computer Graphics*, PP:1–1, 2015.

Jacob C Bridgeman and Christopher T Chubb. Hand-waving and interpretive dance: an introductory course on tensor networks. *Journal of Physics A: Mathematical and Theoretical*, 50(22):223001, may 2017. doi: 10.1088/1751-8121/aa6dc3.

Tom B. Brown, Benjamin Mann, Nick Ryder, and Melanie et al. Subbiah. *Language Models are Few-Shot Learners*. 2020.

David Chiang, Alexander M. Rush, and Boaz Barak. Named Tensor Notation. *CoRR*, abs/2102.13196, 2021.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR*, abs/1810.04805, 2018.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Chapter 2*. MIT Press, Cambridge, MA, USA, 2016. http://www.deeplearningbook.org.

John Hayes and Diana Bajzek. Understanding and Reducing the Knowledge Effect: Implications for Writers. *Written Communication*, 25:104–118, 1 2008.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016. URL http://arxiv.org/abs/1606.08415.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A Survey of Transformers. *CoRR*, abs/2106.04554, 2021.

Roger Penrose. Applications of Negative Dimensional Tensors. *Combinatorial Mathematics and its Applications*, pp. 221–244, 1971.

Paolo Perrone. Markov Categories and Entropy, December 2022.

Mary Phuong and Marcus Hutter. *Formal Algorithms for Transformers*. 2022.

Steven Pinker. *Chapter 2, 3, Sense of style: The thinking person's guide to writing in the 21st century*. Penguin, 2014.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022.

Lee Ross, David Greene, and Pamela House. The "false consensus effect": An egocentric bias in social perception and attribution processes. *Journal of Experimental Social Psychology*, 13(3):279–301, 1977.

Sadoski. Impact of concreteness on comprehensibility, interest. *Journal of Educational Psychology*, 85(2): 291–304, 1993.

Dan Shiebler, Bruno Gavranović, and Paul Wilson. *Category Theory in Machine Learning*. arXiv, 2021.

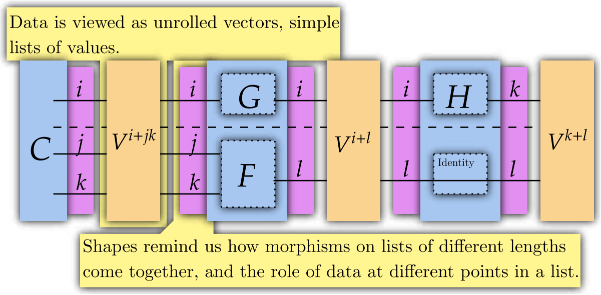
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2017.

Tom Xu and Yoshihiro Maruyama. Neural string diagrams: A universal modelling language for categorical deep learning. 2021. doi: 10.1007/978-3-030-93758-4_32.

A Appendix

Without using category theory, we can establish a formal mathematical justification that ensures neural circuit diagrams are robust by viewing data as unrolled vectors. This gives us confidence that stacked operations have clear, unambiguous mathematical meaning. However, as we are not using category theory, much of the essential structure is lost in this formalism.

In this formalism, we view all data as lists of values of some number field, V, which accepts addition and multiplication. Shapes remind us of the structure of data but do not impose mathematical properties. $V^{4\times 3}$ is the same as V^{12}. When we combine morphisms by stacking or collecting, we create new morphisms with larger input and output vector lengths. The shapes are there to remind us how morphisms have been combined.

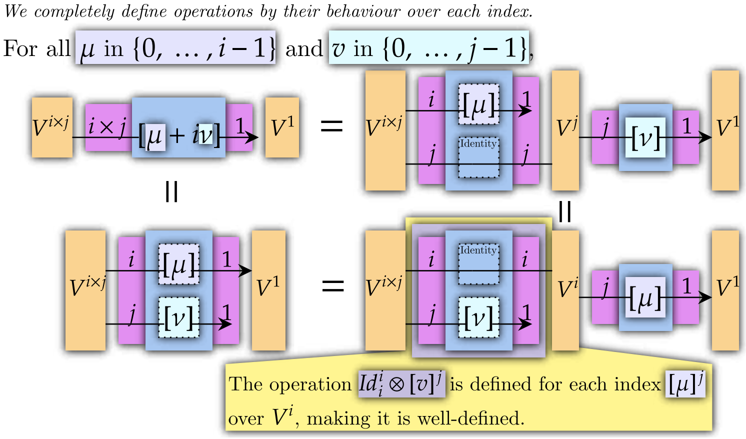


Data is viewed as unrolled vectors, simple lists of values.

Shapes remind us how morphisms on lists of different lengths come together, and the role of data at different points in a list.

Figure 13: In this perspective, shapes remind us of the role that data plays but stacked operations are ultimately functions from one list to another.

The result of accessing each index fully defines data. Because lists $v = \begin{pmatrix}1 & 3 & 2\end{pmatrix}$ and $w = \begin{pmatrix}1 & 3 & 2\end{pmatrix}$ are equal for each index; so they are completely equal for all cases. The same applies for morphisms to lists V^n. If two morphisms are equal for each index, they are equal, and hence we can use one to define the other. For an n-length list V^n, there are n index morphisms $[\mu]^n : V^n \to V$ for $\mu \in \{0, \dots, n-1\}$ which identify each way morphisms to lists of that length can vary.



We completely define operations by their behaviour over each index.

For all μ in $\{0, \dots, i-1\}$ and v in $\{0, \dots, j-1\}$,

The operation $Id_i^i \otimes [v]^j$ is defined for each index $[\mu]^j$ over V^i, making it is well-defined.

Figure 14: Isolating specific axes with indexes follows from the above equalities, which fully define using indexes coupled with identities to access data.

Collecting morphisms $f : V^i \to V^j$ and $g : V^k \to V^l$ gives a morphism $(f \oplus g) : V^{i+k} \to V^{j+l}$ such that the first i elements of the input are fed through f to generate the first j elements of the output, and the remaining input elements are fed to g to give the remaining output elements. This means collected morphisms are precisely defined. This loose definition allows any morphisms to be collected.

Coupling is more restrictive than collecting. For a list $V^{i \times j}$, we define coupled indexes $[\mu]^i \otimes [\nu]^j : V^{i \times j} \to V^1$ for $\mu \in \{0, \ldots, i-1\}$ and $\nu \in \{0, \ldots, j-1\}$, which are equal to the$[\mu + i\nu]^{i \times j}$ index of $V^{i \times j}$. Morphisms are exactly defined by their behavior concerning every output index, so the diagram in Fig. 14 defines indexes coupled with the identity.

We define both coupling and inner coupling as transforming an operation $F : V^{l+i} \to V^k$ to Coupling(F, l, j) : $V^{l+i \times j} \to V^{k \times j}$ by enforcing the statements in Fig. 15 for each index of j. The equality implies the resulting V^k vectors are equal. To get normal coupling, we set l to zero. Furthermore, we allow vertical reflection in the diagram to define coupling with the identity over k above rather than below.

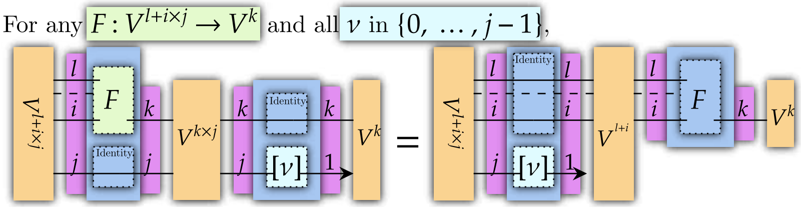


Figure 15: The above diagram completely defines inner coupling. Setting l to zero or vertically reflecting the diagram covers additional cases.

We allow for stacks of more than two morphisms when the order of stacking, $F * (G * H)$ against $(F * G) * H$ with $*$ as coupling or collecting, does not affect the relationship between input and output data. This is always the case for collecting and only sometimes the case for coupling morphisms.