UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

ANALYSIS OF NEURONS IN A TRANSFORMER NEURAL MACHINE TRANSLATION MODEL

ROBERTO RUIZ WILSON

Thesis supervisor
CARLOS ESCOLANO PEINADO (Department of Computer Science)

Thesis co-supervisor
JAVIER FERRANDO MONSONIS (Department of Computer Science)

Degree
Master's Degree in Innovation and Research in Informatics ()

Master's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

24/10/2024

Abstract

This study presents a comprehensive analysis of individual neurons within the final feed-forward network (FFN) layer of a multilingual transformer-based Neural Machine Translation (NMT) model. Our research demonstrates that neurons in this layer are highly interpretable, offering unprecedented insights into the internal mechanisms of these complex systems. By examining neuron activations and weights across various language pairs, we uncover specialized neurons responsible for specific linguistic functions, from lexical choice to syntactic structure.

Our methodology combines systematic neuron weight analysis with visualization techniques, revealing patterns in how the model translates text across multiple languages. Notably, we find that certain neurons play crucial roles in influencing output language choice, and that specific neurons are tied to specific output languages. We find that the model even clusters very similar languages as in the case of Serbo-Croatian, even though they are labeled separately.

Acknowledgements

This work results from Javier Ferrando's vision and persistence in supporting me throughout this process. This would not exist without him.

I would also like to thank my supervisor Carlos Escolano for his patience, support, and for making this possible.

I am deeply indebted to my family and friends for their support, and understanding throughout my academic journey. Their encouragement has been what kept me going.

Lastly, I would like to thank all the researchers whose work has inspired and informed this study.

Contents

List of Figures

1 Introduction

Neural machine translation (NMT) has significantly advanced the field of natural language processing, with transformer-based models demonstrating exceptional proficiency in translating text across multiple languages. While the overall performance of these models is well-documented, there remains a critical gap in our understanding of their internal operations. This research thesis aims to address this gap by conducting an in-depth analysis of individual neurons within a multilingual transformer-based NMT model, specifically focusing on the final feed-forward network (FFN) layer.

Our investigation centers on interpreting the weights within the last FFN layer, an often-overlooked component of the transformer architecture that significantly influences the final stages of translation. By analyzing the weights and contributions of these neurons, we aim to uncover the underlying mechanisms that enable these models to perform complex linguistic tasks across multiple languages.

We employ visualization techniques to map the relationships between neuron weights and specific linguistic phenomena. This analysis may reveal specialized neurons that focus on particular aspects of translation, such as handling cross-lingual transfer or clustering across language families.

The insights gained from this research have the potential to inform the design of more efficient and effective multilingual NMT architectures. By identifying critical neurons and understanding their roles across languages, we can potentially optimize network structures, leading to more streamlined and powerful translation models.

1.1 Objectives

Primarily, we aim to analyze the weights of individual neurons in the final FFN layer of a multilingual transformer-based NMT model. This analysis will allow us to identify

and characterize neurons that play crucial roles in specific aspects of the translation process, such as lexical choice, syntactic structure, or cross-lingual transfer.

Ultimately, our research aims to contribute to the broader understanding of interpretability in multilingual NMT models. By shedding light on the inner workings of these systems, we hope to contribute to the creation of more transparent, efficient, and linguistically informed NMT systems.

By achieving these objectives, we aim to bridge the gap between the impressive performance of multilingual NMT models and our understanding of their internal mechanisms. The results of this study are expected to contribute insights into the fields of machine translation, interpretable AI, and neural network analysis, potentially leading to more transparent, efficient, and linguistically informed NMT systems that perform well across a diverse range of languages, including those with limited resources.

2 Theoretical Framework

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational models designed to process information and learn patterns from data. These networks consist of interconnected units, called artificial neurons or nodes, that work collectively to solve complex problems. ANNs excel at tasks such as classification, regression, and decision-making, making them valuable tools in various fields including computer vision, natural language processing, and data analysis [15, 14]. The primary strength of ANNs lies in their ability to learn from examples. Rather than being explicitly programmed with rules, these networks adjust their internal parameters based on the data they encounter. This adaptability enables ANNs to address problems that are challenging to solve with traditional, rule-based programming approaches.

2.1.1 Structure of Artificial Neural Networks

The fundamental unit of an artificial neural network is the artificial neuron, also termed a node. An artificial neuron is a computational unit that processes information. The functioning of an artificial neuron can be described through its components and operations [26, 27].

An artificial neuron receives one or more inputs. These inputs can originate from external data, in the case of neurons in the input layer, or from other neurons in the network. Each input represents a piece of information that the neuron will process.

Associated with each input to a neuron is a weight. A weight is a numerical value that represents the importance or strength of that particular input. The neuron learns these weights during the training process. Weights with larger absolute values indicate that the corresponding input has a stronger influence on the neuron's output.

The neuron combines its inputs by calculating a weighted sum. This operation involves multiplying each input by its corresponding weight and then summing all these products. Mathematically, for inputs $x_1, x_2, ..., x_n$ and weights $w_1, w_2, ..., w_n$, the weighted sum is expressed as:

$$\text{weighted sum} = w_1x_1 + w_2x_2 + ... + w_nx_n$$

In addition to the weighted sum, a neuron typically incorporates a bias term. The bias is an additional parameter that allows the neuron to shift its output. It functions as a constant input that is always active. The bias aids the neuron in learning more effectively by enabling it to shift its activation function. Adding the bias to the weighted sum yields:

$$z = w_1x_1 + w_2x_2 + ... + w_nx_n + b$$

where b represents the bias.

After computing the weighted sum plus bias, the neuron applies an activation function to this value. The activation function introduces non-linearity into the neuron's output, which is crucial for the network's ability to learn complex patterns. The activation function determines whether and to what extent the neuron should transmit information. If we denote the activation function as f, then the neuron's output is expressed as:

$$y = f(z) = f(w_1x_1 + w_2x_2 + ... + w_nx_n + b)$$

2.1.2 Activation Functions

The result of the activation function is the neuron's output. This output can serve as an input to other neurons in the network, or if the neuron is in the output layer,

it might represent the final result of the network's computation.

Activation functions are a crucial component of artificial neurons, introducing non-linearity into the neural network's computations. This non-linearity allows neural networks to learn and approximate complex functions, which is essential for tasks such as pattern recognition and decision making [24].

The activation function takes the weighted sum of inputs plus the bias as its argument and produces the neuron's output. Several types of activation functions exist, each with its own characteristics and use cases. Three commonly used activation functions are the sigmoid function, the hyperbolic tangent (tanh) function, and the Rectified Linear Unit (ReLU) function.

The sigmoid function, defined as $f(x) = \frac{1}{1+e^{-x}}$, maps its input to a value between 0 and 1. This property makes it useful for models where the output needs to be interpreted as a probability. However, the sigmoid function suffers from the vanishing gradient problem for very large or very small inputs, which can slow down learning in deep networks [13].

The hyperbolic tangent (tanh) function, defined as $f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$, is similar to the sigmoid but maps inputs to values between -1 and 1. This centered nature of tanh often makes it preferable to the sigmoid in practice. Like the sigmoid, tanh also faces the vanishing gradient problem for extreme input values.

The Rectified Linear Unit (ReLU) function, defined as $f(x) = max(0, x)$ is the most frequently used activation function. ReLU simply returns 0 for any negative input and the input value itself for any positive input. This function does not saturate for positive values, which helps mitigate the vanishing gradient problem. ReLU is computationally efficient as it involves simpler mathematical operations. However, it can suffer from the "dying ReLU" problem, where neurons can get stuck in a state where they never activate [20].

2.1.3 Neural Network Architecture

The choice of activation function can significantly impact a neural network's performance and training dynamics. Different activation functions may be more suitable for different layers of the network or for different types of tasks. In practice, ReLU and its variants are often used in hidden layers due to their computational efficiency and effectiveness in deep networks, while functions like sigmoid or softmax are commonly used in output layers for classification tasks [24].

The architecture of a neural network refers how its constituent neurons are arranged into a coherent structure. This structure typically consists of multiple layers of neurons, each performing specific computations on the input data. The most common type of neural network architecture is the feedforward network, where information flows in one direction from the input layer through one or more hidden layers to the output layer [27].

The input layer of a neural network serves as the interface between the raw data and the network. Each neuron in this layer represents a feature of the input data. For instance, in an image recognition task, each input neuron might correspond to a pixel in the image. The input layer does not perform any computations; it merely passes the data to the subsequent layer.

Hidden layers form the core of the neural network's computational capacity. These layers are termed "hidden" because they are not directly observable from the network's inputs or outputs. Each hidden layer consists of multiple neurons, with each neuron connected to all neurons in the previous and subsequent layers in a fully connected architecture. The number of hidden layers and the number of neurons in each layer are key architectural decisions that influence the network's capacity to learn complex patterns [3].

The output layer produces the final result of the network's computations. The

structure of this layer depends on the specific task the network is designed to perform. For a binary classification task, the output layer might consist of a single neuron with a sigmoid activation function. For multi-class classification, it might have multiple neurons, one for each class, often using a softmax activation function to produce a probability distribution over the classes [14].

The connections between neurons in adjacent layers are represented by weights, which the network learns during training. These weights determine the strength of the connections and, consequently, the influence of each neuron on the neurons in the subsequent layer. The process of training adjusts these weights to minimize the difference between the network's predictions and the true outputs for a given set of inputs.

The depth of a neural network, referring to the number of hidden layers, has a significant impact on its learning capacity. Deep neural networks, with many hidden layers, can learn hierarchical representations of the data, with each successive layer capturing increasingly abstract features. This hierarchical learning is particularly effective for complex tasks such as image and speech recognition [19].

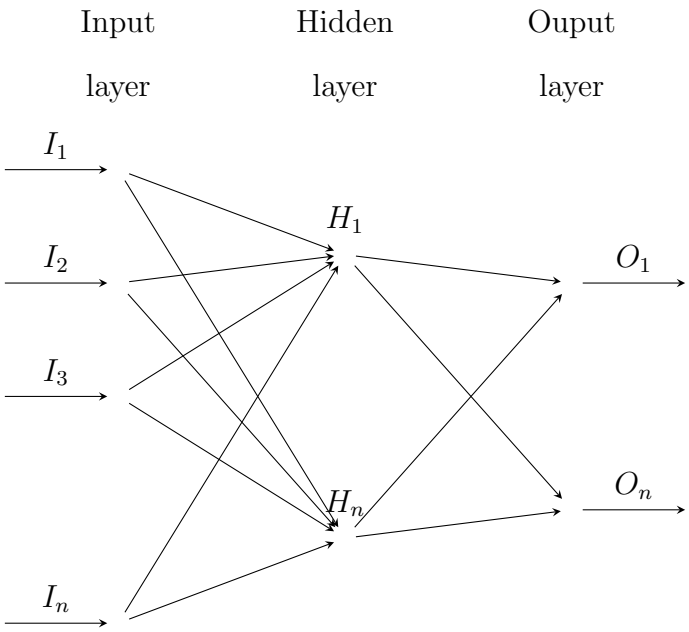

Figure 1: Fully-connected feedforward network

2.2 Computational Process in Neural Networks

2.2.1 Forward Pass

The forward pass, also known as forward propagation, is the process by which a neural network computes its output (or prediction) for a given input. This process involves the sequential application of the operations we've discussed in the previous sections, moving from the input layer through the hidden layers to the output layer [14]. When an input is presented to the network, it is processed sequentially through each layer. In each neuron, the weighted sum of inputs is calculated and then passed through an activation function.

The computation begins at the input layer, where the feature values of the input data are presented to the network. For each subsequent layer, including hidden layers and the output layer, the network computes the weighted sum of inputs from the previous layer, adds a bias term, and then applies an activation function to produce

that layer's output.

Mathematically, for a given layer l, the output of each neuron j in that layer can be expressed as:

$$a_j^l = f\left(\sum_i w_{ji}^l a_i^{l-1} + b_j^l\right)$$

Where a_j^l is the activation (output) of neuron j in layer l, w_{ji}^l is the weight connecting neuron i in layer $l-1$ to neuron j in layer l, a_i^{l-1} is the activation of neuron i in the previous layer, b_j^l is the bias of neuron j in layer l, and f is the activation function [27].

This computation is performed for each neuron in each layer, propagating the input signal forward through the network. The specific nature of the computation in the output layer depends on the task the network is designed to perform. For a regression task, the output might be a single continuous value. For a binary classification task, it might be a single value between 0 and 1, representing the probability of the positive class. For multi-class classification, the output layer typically produces a probability distribution over all possible classes [5].

In the case of multi-class classification, and for the model we'll be examining, the softmax function is often used in the output layer. This function takes a vector of arbitrary real-valued scores and transforms it into a vector of values between 0 and 1 that sum to 1, which can be interpreted as probabilities. The softmax function for the j-th class is defined as:

$$\text{softmax}(z)_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Where z_j is the input to the softmax function for class j, and the sum in the denominator is over all classes [6].

The forward pass is not only crucial for making predictions with a trained network

but also forms the first step in the training process. During training, the outputs computed by the forward pass are compared to the true targets to compute the loss, which is then used to update the network's parameters through backpropagation.

2.2.2 Calculating Loss

Following the forward pass, the neural network's performance is evaluated by computing a loss function. In the context of translation models and other sequence-to-sequence tasks, the most commonly used loss function is the cross-entropy loss, also known as log-loss [14].

Cross-entropy loss quantifies the difference between the predicted probability distribution and the true distribution of the target sequence. For a single token in the output sequence, the cross-entropy loss is defined as:

$$L = -\sum_{j=1}^{C} y_j \log(\hat{y}_j)$$

Where C is the size of the vocabulary (i.e., the number of possible tokens), y_j is a binary indicator of whether token j is the correct next token in the sequence (1 for the correct token, 0 for all others), and $\hat{y}_j$ is the predicted probability of token j being the next token in the sequence [30].

In practice, for a sequence of tokens, the loss is computed as the average cross-entropy loss over all tokens in the sequence:

$$L_{\text{sequence}} = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{C} y_{j,t} \log(\hat{y}_{j,t})$$

Where T is the length of the sequence, and $y_{j,t}$ and $\hat{y}_{j,t}$ are the true and predicted probabilities for token j at position t in the sequence, respectively [2].

The use of cross-entropy loss in translation models is particularly appropriate

because:

- It naturally handles the multi-class nature of the problem, where each token in the output sequence is chosen from a large vocabulary.
- It provides a probabilistic interpretation of the model's outputs, which is useful in tasks like beam search during inference.
- Minimizing cross-entropy loss is equivalent to maximizing the likelihood of the correct translations given the input sequences, which aligns with the goal of translation models [14].

During training, the loss is typically averaged over a batch of input-output sequence pairs, providing a more stable estimate of the model's performance. This average loss serves as the objective function that the training process aims to minimize.

The computed loss plays a crucial role in the subsequent step of the training process, backpropagation, where it is used to compute gradients and update the network's parameters. The goal of training is to find the set of parameters that minimizes the expected loss over the entire dataset of translation pairs.

2.2.3 Backpropagation

Backpropagation is the fundamental algorithm used to train neural networks, including those used in transformer-based translation models. This algorithm computes the gradients of the loss function with respect to the network's parameters, which are then used to update these parameters in a direction that reduces the loss [27].

The backpropagation process begins with the computed loss from the output layer and works backwards through the network, layer by layer. For each parameter in the network, backpropagation calculates how a small change in that parameter would

affect the overall loss. This calculation applies the chain rule of calculus repeatedly to compute partial derivatives [14].

In a translation model, for a loss L corresponding to a single token in the output sequence, the gradient of this loss with respect to a weight w in the network is computed as:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Where a is the activation of a neuron, z is its weighted input, and w is a weight parameter. This computation is performed for every weight in the network, moving from the output layer back to the input layer [19].

After gradient computation, the network's parameters are updated using an optimization algorithm. The basic form of this update is gradient descent:

$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial w}$$

Where η is the learning rate, a hyperparameter that controls the size of the update steps. More sophisticated optimization algorithms such as Adam are often used in practice, which adapt the learning rate for each parameter based on the statistics of recent gradients [17].

In translation models, where the loss is computed over sequences of tokens, the parameter updates are typically performed after accumulating gradients over a batch of input-output sequence pairs. This batch processing stabilizes the learning process and allows for efficient computation on modern hardware.

The backpropagation and update process is repeated iteratively over the entire training dataset, with each complete pass over the dataset known as an epoch. Over multiple epochs, the network adjusts its parameters to minimize the loss function, improving its ability to generate accurate translations.

Conceptually, backpropagation can be understood as a backward flow of error information through the network. Starting from the output layer, the algorithm computes how much each neuron contributed to the overall error. This error is then propagated backwards, layer by layer, allowing the computation of gradients for all parameters in the network. Once the gradients are computed, the network's parameters are updated in a direction that reduces the loss.

2.3 Byte Pair Encoding (BPE)

Byte Pair Encoding (BPE) is a subword tokenization algorithm used in natural language processing for vocabulary construction. It operates on the principle of iteratively merging the most frequent pair of bytes or characters in a corpus. BPE serves as a data compression technique and a method to handle out-of-vocabulary words in machine learning models [29]. The algorithm initializes by splitting words into individual characters and then progressively builds a vocabulary of subword units. It iteratively identifies and merges the most frequent adjacent pair of symbols, creating a new symbol. This process continues until a predefined vocabulary size is reached or no more merges are possible [11].

The BPE algorithm begins by initializing the vocabulary with individual characters from the corpus. It then counts the frequency of adjacent symbol pairs in the corpus. The most frequent pair is merged and added to the vocabulary. This process is repeated until the desired vocabulary size is reached or no more merges are possible. This iterative approach allows BPE to capture common subword units and adapt to the specific characteristics of the training corpus [18].

Formally, let V be the vocabulary and C be the corpus. The BPE algorithm can be expressed as a series of operations on V and C. In each iteration, the algorithm identifies the pair (x, y) that maximizes the frequency count in C. If this count is

zero, the algorithm terminates. Otherwise, it adds the merged pair xy to V and replaces all occurrences of xy in C with the new token. This process effectively builds a vocabulary that balances frequency and granularity of subword units.

Algorithm 1 Byte Pair Encoding

$V \leftarrow$ set of unique characters in C
while $|V| <$ desired vocabulary size **do**
  $(x, y) \leftarrow \arg\max_{(a,b) \in V \times V} \text{count}(ab, C)$
  if $\text{count}(xy, C) = 0$ **then**
    break
  end if
  $V \leftarrow V \cup xy$
  Replace all occurrences of xy in C with the new token
end while

BPE provides several advantages in natural language processing tasks. It creates a balance between character-level and word-level representations, allowing for effective handling of morphological variants and rare words [29]. It generates a fixed-size vocabulary, which is beneficial for neural network models with fixed-size input and output layers. Additionally, it enables the model to handle out-of-vocabulary words by decomposing them into known subword units, improving the model's ability to process unseen or rare words [32].

The choice of vocabulary size in BPE presents a trade-off between the granularity of subword units and the computational efficiency of the model. A larger vocabulary allows for more whole-word representations but increases model complexity, while a smaller vocabulary results in more subword segmentations but may lose some semantic information. This trade-off is an important consideration in the application of BPE to various natural language processing tasks [18].

2.4 Embeddings

Embeddings are dense vector representations of discrete tokens, crucial for capturing semantic relationships in the input data. Embeddings address several technical challenges in language processing: they reduce the dimensionality of token representation, capture semantic relationships between tokens, and provide a continuous representation that allows for effective gradient-based learning [4].

The primary function of embeddings is to transform high-dimensional, sparse one-hot vectors into lower-dimensional, dense vectors. This transformation typically reduces the dimensionality from the size of the vocabulary (often tens or hundreds of thousands) to a much smaller dimension, commonly ranging from 100 to 1000. This reduction in dimensionality improves computational efficiency and helps mitigate the curse of dimensionality [21].

In the embedding space, tokens with similar semantic or syntactic properties tend to have similar vector representations. This property allows the model to generalize across similar words and capture analogical relationships. The geometric properties of the embedding space often reflect linguistic relationships, which can be quantified using vector operations [22].

Embeddings facilitate transfer learning in natural language processing tasks. Pre-trained embeddings, learned from large corpora, can be used as initial representations for various downstream tasks, potentially improving performance and reducing the amount of task-specific training data required [25].

In the transformer architecture [31], token embeddings form the initial representation of the input sequence. Let V be the vocabulary size and d be the embedding dimension.[31] The token embedding matrix $E \in \mathbb{R}^{V \times d}$ is a learned parameter of the

model. For a token t, its embedding e_t is defined as:

$$e_t = E[t]$$

The embedding matrix E is typically initialized randomly and updated during the training process through backpropagation. The update rule for the embedding matrix can be expressed as:

$$E_{t+1} = E_t - \eta \frac{\partial \mathcal{L}}{\partial E_t}$$

Where E_t is the embedding matrix at time step t, η is the learning rate, and $\mathcal{L}$ is the loss function.

Similarity between token embeddings can be measured using the cosine similarity:

$$\text{similarity}(e_i, e_j) = \frac{e_i \cdot e_j}{|e_i||e_j|}$$

Where e_i and e_j are embeddings for tokens i and j, respectively.

These embeddings serve as the initial representation of the input sequence in the transformer model, providing a foundation for subsequent processing layers.

2.5 Transformer

The Transformer is a neural network architecture designed to process sequential data, such as text [31]. At its core, it consists of an encoder and a decoder, both composed of stacked layers. The key innovation of the Transformer lies in its reliance on attention mechanisms, particularly self-attention, to process input sequences.

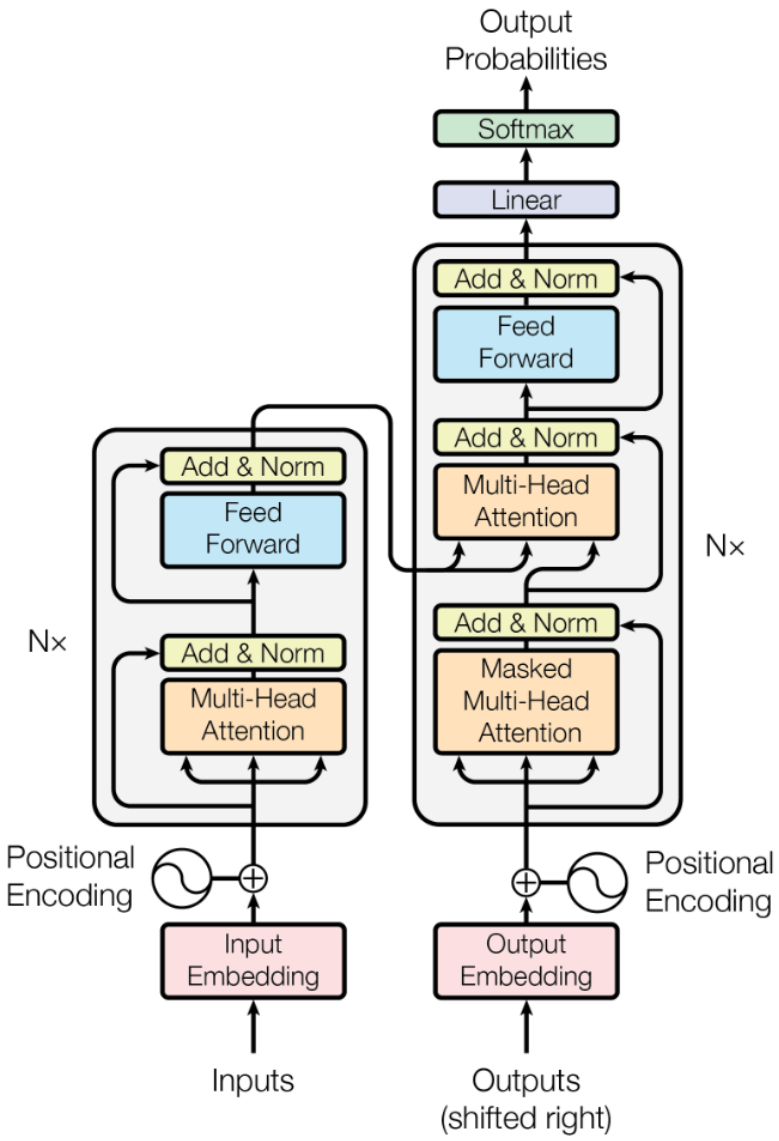

Figure 2: Transformer Architecture Source [31]

Unlike recurrent neural networks (RNNs), that process data sequentially, the Transformer considers the entire input sequence simultaneously. This parallel processing approach allows for more efficient computation and does a better job capturing dependencies between different tokens in the input data, regardless of the distance between elements in the sequence.

2.5.1 Self-Attention Mechanism

The self-attention mechanism is a fundamental component of the Transformer architecture, enabling the model to weigh the importance of different parts of the input when processing each element of a sequence. This mechanism allows the Transformer to capture complex relationships and dependencies within the data, irrespective of the distance between elements in the sequence [31].

To understand self-attention, we can think about what "attention" means in the general sense. Essentially, attention in neural networks is a way for the model to selectively focus on certain parts of the input data. It's analogous to how humans focus on specific words or phrases when reading a sentence to parse its meaning. The name self-attention is due to the model attending to its own input sequence.

The self-attention mechanism operates on three projections of the input sequence: Query (Q), Key (K), and Value (V) matrices. These matrices are created by multiplying the input embeddings with learned weight matrices. Each row in these matrices corresponds to a token in the input sequence. The dimensions of these matrices are determined by the model's hyperparameters, typically with shapes $(sequence\_length, d_{model})$, where d_{model} is the dimensionality of the model's internal representations.

The Query matrix represents the current token's "questions" or points of interest. The Key matrix holds the "answers" or characteristics of all tokens that could be matched against these questions. The Value matrix contains the actual information to be aggregated. This Query-Key-Value concept is inspired by information retrieval systems, where a query is used to find relevant keys, which then point to the desired values.

The attention mechanism computes how much each token in the sequence should attend to every other token. This is done by comparing each query vector with all

key vectors:

$$\text{Attention Scores} = QK^T \tag{1}$$

These scores indicate how relevant each token is to every other token. Higher scores suggest stronger relationships or dependencies between tokens. The scores are then scaled and normalized using a softmax function:

$$\text{Attention Weights} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \tag{2}$$

Here, d_k is the dimensionality of the key vectors, and the scaling factor $\sqrt{d_k}$ helps maintain stable gradients during training. Finally, these attention weights are used to compute a weighted sum of the value vectors, producing the output of the self-attention operation:

$$\text{Output} = \text{Attention Weights} \cdot V \tag{3}$$

This process allows each token to gather information from all other tokens in the sequence, with the degree of information flow determined by the attention weights. Tokens that are more relevant to each other will have higher attention weights, allowing for stronger information exchange.

The flexibility of the self-attention mechanism allows it to capture different kinds of relationships and hidden structure within the data. In Natural Language Processing, it can find on semantically related concepts, syntactic structures, and long-distance references.

2.5.2 Multi-Head Attention

Multi-head attention is an extension of the self-attention mechanism that enhances the Transformer's ability to focus on different aspects of the input sequence simultaneously. This approach allows the model to capture various types of relationships

and dependencies within the data more effectively than a single attention mechanism [31].

The main intuition for multi-head attention is to perform the attention operation multiple times in parallel, with each iteration referred to as an "attention head." Each head learns its own set of Query (Q), Key (K), and Value (V) projections, allowing it to focus on different aspects of the input sequence. The outputs from all heads are then concatenated and linearly transformed to produce the final result.

Mathematically, multi-head attention can be expressed as follows:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{4}$$

where each head is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{5}$$

Here, W_i^Q, W_i^K, and W_i^V are learned parameter matrices for the i-th head, and W^O is a learned parameter matrix that combines the outputs of all heads.

The use of multiple attention heads provides several advantages. Firstly, it allows the model to attend to information from different representation subspaces. For instance, one head might focus on local syntactic relationships, while another captures long-range semantic dependencies. This multi-faceted approach enriches the model's understanding of the input sequence.

Secondly, multi-head attention introduces an element of ensemble learning within the attention mechanism itself. By combining the outputs of multiple attention operations, the model can potentially achieve more robust and stable representations, as different heads may capture complementary aspects of the input.

The number of attention heads is a hyperparameter that can be tuned based on

the specific task and dataset. Common configurations in practice range from 8 to 16 heads, but the optimal number may vary depending on the complexity of the task and the size of the model.

The multi-head attention mechanism is used in various components of the Transformer architecture, including the self-attention layers in both the encoder and decoder, as well as in the encoder-decoder attention layer.

2.5.3 Encoder-Decoder Structure

The encoder-decoder structure consists of two main components: the encoder and the decoder [31]. The encoder takes the input sequence and processes it to create a condensed representation, often called the "context" or "encoding." It tries to understand and capture the "essence" of the input. It is made of a stack of identical layers, with each one containing a multi-head self-attention mechanism, and a feed-forward neural network. In the encoder's self-attention layers, each position attends to all positions in the input sequence.

Intuitively, the encoder can be thought of as someone reading a text. It reads the entire input sequence, understanding not only the individual elements but also how they relate to each other. By allowing each position to attend to all others, it can capture complex relationships and dependencies, regardless of their distance in the sequence.

The decoder takes the encoding produced by the encoder and generates the output sequence. It essentially "decodes" the information captured by the encoder to produce the desired output, also using any previously generated outputs. It also consists of a stack of identical layers. Each decoder layer contains three sublayers: a masked multi-head self-attention mechanism, a multi-head attention over the encoder output, and a position-wise feed-forward network.

The decoder is like to a writer who has is given notes by the encoder and has to produce a new text, word by word. While it generates each element of the output, it refers back to these notes, focusing (attending) on the most relevant parts.

Another interpretation for the interaction between encoder and decoder is as a question-answering process. The decoder asks questions (queries) about the input, and the encoder provides relevant information (keys and values) that could answer these questions.

2.5.4 Layer Normalization

Layer normalization normalizes the inputs across the features [1]. It is applied after each sub-layer in the encoder and decoder. For an input vector x, layer normalization is defined as:

$$\text{LayerNorm}(x) = \alpha \odot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{6}$$

With μ and σ being the mean and standard deviation of the input features, α and β are learnable parameters, and ϵ is a small constant for numerical stability.

Layer normalization helps in two ways. It reduces the internal covariate drift, which is the change in the distribution of network activations due to the change in network parameters during training. It also allows each layer to learn more independently of other layers.

Residual connections, also known as skip connections, allow the model to bypass the sub-layers. Each sub-layer output is added to its input and then normalized:

$$x = \text{LayerNorm}(x + \text{Sublayer}(x)) \tag{7}$$

2.5.5 Masked Attention

Masked attention prevents the model from attending to future tokens during training and inference [31]. This mechanism ensures that predictions for a given position can only depend on known outputs at earlier positions, preventing data leakages.

The masked attention is implemented in the decoder's self-attention layers. A general form of the attention mechanism, which includes the possibility of masking, can be expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T + M}{\sqrt{d_k}}\right)V \tag{8}$$

Where M is an optional mask. The specific form of the mask can be implemented as[28]:

$$M_{ij} = \begin{cases} 0 & \text{if } i \geq j \\ -\infty & \text{if } i < j \end{cases} \tag{9}$$

The token in position j can be "looked at" by the token in position i. If $j > i$ then it's a "future" token and is masked out. The mask workes by assigning a weight of -inf which the softmax will output as a probability of 0. The matrix pattern can be seen in 3

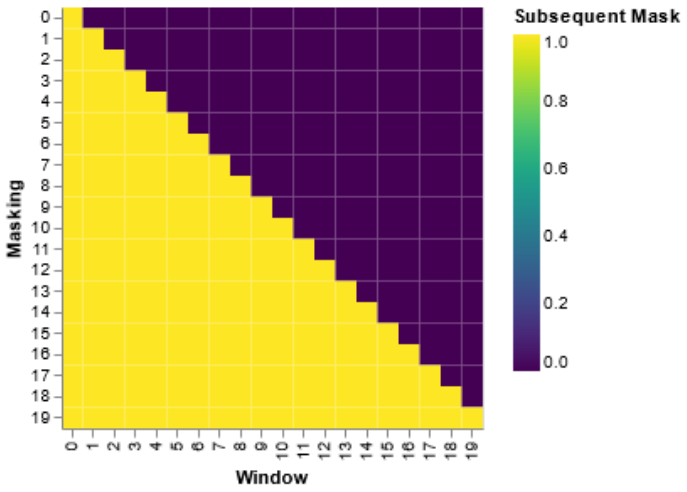

Figure 3: Binary mask pattern, in practice values are implemented as 0 or -inf [28]

2.6 Information Flow and Residual Stream

The transformer architecture consists of a sequence of components beginning with token embedding, followed by multiple "residual blocks," and concluding with token unembedding [9]. Each residual block comprises an attention layer followed by an MLP layer. These layers interact with a "residual stream," which serves as the central communication channel in the transformer's design. The residual stream functions as follows:

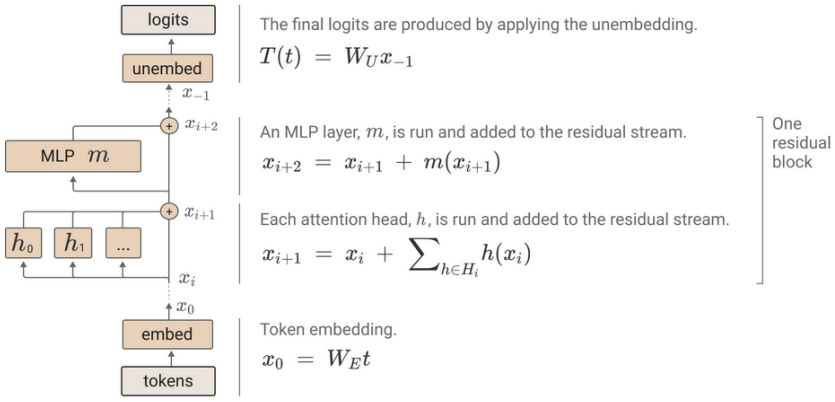

Figure 4: Information flow through the residual stream [9]

Attention and MLP layers extract information from the stream via linear projections (reading). These layers then contribute their processed results back to the stream through additional linear projections (writing). Within each attention layer, multiple heads operate in parallel to process information.

Mathematically, the residual stream at any point is defined as the sum of the original embedding and the outputs of all preceding layers. It can be conceptualized as a communication channel, facilitating information flow between layers without performing processing itself.

Each layer begins by applying an arbitrary linear transformation to read from the stream. It then applies another linear transformation before adding its output back to the stream.

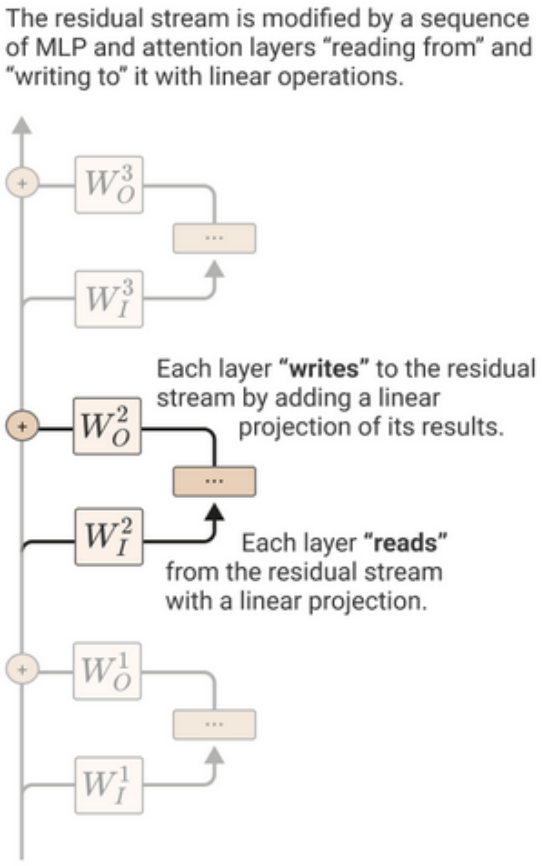

Figure 5: Effect of layers on the residual stream [9]

The residual stream is a continuous flow of information through the transformer. It starts with the initial token embeddings and passes through each layer of the model. As the information moves through each layer, new data is added to the stream. This additive property is the reason for the term "residual" - each layer contributes to the existing information rather than replacing it entirely.

An important characteristic of the residual stream is that it uses linear operations. Information is added or subtracted in the stream, without complex non-linear transformations. This allows for analysis of information flow through the model, which is the basis of this work.

2.7 Feed-Forward Layers as Key-Value Memories

Feed-forward layers in transformer models function as key-value memories [12]. This is a novel way of interpreting how these layers process and store information.

Recall that a feed-forward layer consists of two linear transformations with a non-linear activation function in between, represented as $FF(x) = f(xK^T)V$, where x is the input vector, K and V are parameter matrices, and f is a non-linear activation function, usually ReLU. We showed an equivalent formulation in 2.2.1 expressing it as individual vectors.

In the key-value memory framework, each row of the K matrix represents a key vector, corresponding to specific patterns in the input data. When an input x is processed, its dot product with each "key" determines how strongly that key is activated. Similarly, each row of the V matrix represents a "value" vector, which can be interpreted as distributions over the output vocabulary. These values encode information about likely outputs associated with the patterns detected by the keys. Our analysis will try to find what information is stored in the "value" matrix of the last layer of our model.

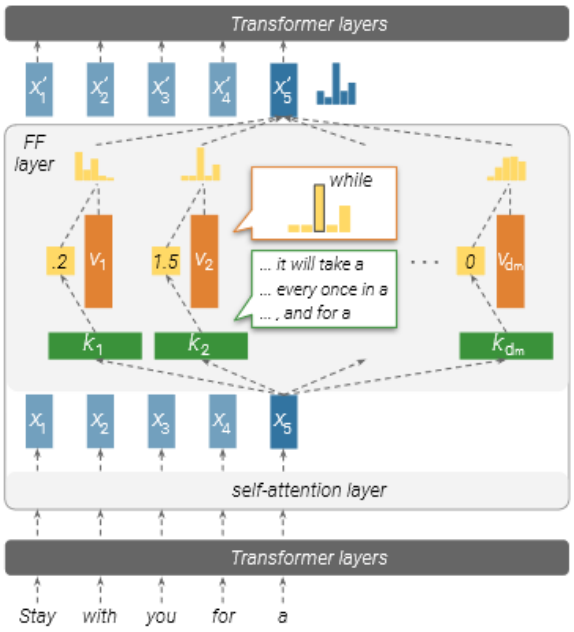

Figure 6: FFN as key-value store. Values of K_2 produce related concepts when combined with the input, when dot product with V_2 is calculated, an appropriate continuation token is given high weight [12]

The input x is compared against all keys simultaneously via matrix multiplication, producing a vector of "memory coefficients" that indicate how strongly each key matches the input. Next, the activation function is applied to these coefficients, introducing sparsity and effectively selecting which memories are relevant for the given input.6 shows this interaction.

These activated coefficients are then used to weight the values through another matrix multiplication. The result is a weighted sum of the selected value vectors, which can be interpreted as a composition of the activated memories, each contributing information about likely output tokens.

3 Methodology

We will consider the top section of the transformer model, and for simplicity we will ignore the bias term. After the output of the last layer is calculated, it is multiplied by the unembedding matrix and is sent to a softmax function to calculate the output probability over the token vocabulary. The output of the last layer is being converted twice, starting from a vector in the embedding space and resulting in a probability distribution; and eventually in a concrete prediction. That is:

1. In the last Layer L, $x_i^{(L-1)}$ is multipled by the last Key matrix $W_{in}^{(L)}$ and the activation function is applied.

2. The result of the previous step is multiplied by the last Value matrix $W_{out}^{(L)}$.

3. Layer output $W_{out}^{(L)}$ is added to the residual stream and multiplied by unembedding matrix W_U. The resulting matrix contains vectors that assign a weight to each token.

4. A softmax is calculated, converting the vector vectors of weights to vectors of probabilities, such that tokens that are assigned large weights will produce large probabilities.

5. Sampling is done on the probability distribution from the previous step to produce a prediction.

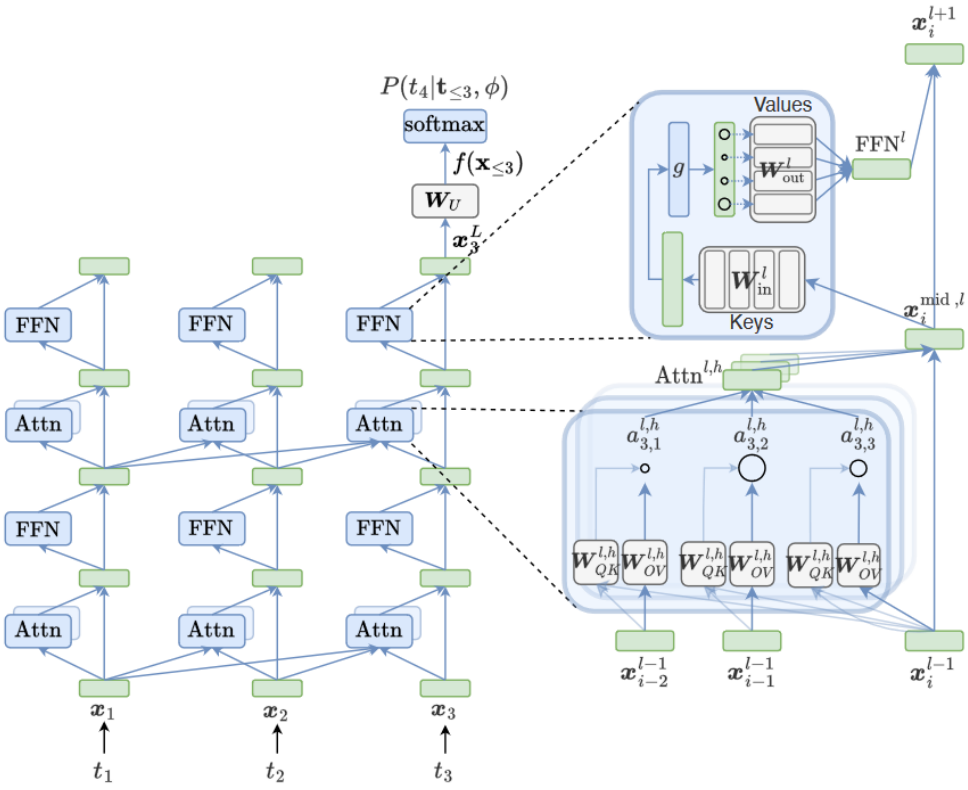

Figure 7: Unrolled transformer, we focus on the top layer [10]

Let g be the result of a specific vector from step 1 in the previous explanation. The computation of the probability distribution can be expressed by:

$$softmax\left(\left(gW_{out}^{(L)} + x_i^{(L-1)}\right)W_U\right)$$

Let us focus on the input to the softmax function. It is a vector that contains a weight for each possible output token. Tokens with high weight will have high probability of being selected as output. Because of linearity, that input is equivalent to:

$$\left(gW_{out}^{(L)}W_U + x_i^{(L-1)}W_U\right)$$

We have split the output weights into two parts: one comes from the residual

stream and the other comes from the last layer of the model. As in the "Expansion Trick [9]. We interpret this as the layer having an effect on the residual stream, as in all layers of the model. What is special about the last layer is since the output gets multiplied by the unembedding matrix W_U, we can directly interpret the weight matrix $W_{out}^{(L)}$ of that last layer. Because of associativity, g gets multiplied by the matrix $\left(W_{out}^{(L)} W_U\right)$, this matrix has vectors in the token output space instead of the embedding space. So we can directly see which specific tokens it affects.

Consider the neuron i such that we have the weight vector $w = \left(W_{out}^{(L)} W_U\right)_i$. That vector will have a weight value for each token in the vocabulary, it will be added to the corresponding i neuron in the residual stream. The result will be the weight that are passed to softmax. This means w will have an effect on the final prediction that can be directly observed. For a token t, if the weight w_t is high, then the total weight will be increased and consequently it's more likely to be selected. If it's very negative, then it will "pull down" its corresponding value and make it less likely to be selected.

We will analyze the last Value vector $W_o^{(L)} ut$ on a state-of-the art multilingual machine translation model. We will look for patterns that can be interpreted in order to learn what the model is doing. We want to find if there is any linguistic structure in our neurons to understand the model.

3.1 Probabilistic Analysis

Once we have our weight neurons, we will try to find if they are associated with a specific output language. We start by finding the probabilistic association of a neuron i to language l.

Let $P(l|t)$ be the probability that token t is of language l, and let $t_1^{(i)}, t_2^{(i)}, ...t_k^{(i)}$ be the k tokens with the largest weight in our neuron i. Intuitively, a high value of $P(l|t)$ means that it's highly likely that token t is of language l. If the neuron contains a

high proportion of high-weight tokens of language l, then that neuron is "pushing" the final output to be output to be of language l. So we define the probabilistic association of neuron i to language l as:

$$\sum_{j<k} P(l|t_j^{(i)})$$

Conversely, the tokens with the lowest assigned weight are also significant. Low weights have negative value, so they "pull away" the final output from the given language.

For our analysis we will consider both meanings of "top k": the k values with the highest and the lowest weight. If our intuition that both are significant is correct, we would see patterns emerge according to both groupings.

3.1.1 Entropy

Entropy quantifies the uncertainty or average information content in a random variable.[8] For a discrete random variable X with possible values $x_1, ..., x_n$ and probability mass function $P(X)$, the entropy $H(X)$ is defined as:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$$

The negative sign ensures that entropy remains non-negative. The choice of logarithm base determines the units of entropy, with base 2 yielding bits and the natural logarithm yielding nats.

The entropy of a random variable equals zero if and only if the variable is deterministic. For a given number of outcomes n, the entropy reaches its maximum value when all outcomes are equally likely:

$$H(X)_{max} = \log_2(n)$$

Less probable events contribute more information when they occur. A uniform distribution, representing maximum uncertainty, has the highest entropy for a given number of outcomes. As the distribution becomes more concentrated on certain outcomes, the entropy decreases, reflecting increased certainty.

We take advantage of this fact to find neurons that carry linguistic structure. A neuron that has no linguistic structure would assign high weights to tokens of seemingly random languages. Its top tokens would come from different languages and the probabilistic association of that neuron would be close to uniform.

A neuron with linguistic structure would have tokens concentrated on a few languages. The entropy would be low. We would be more certain if we were to take a random token and guess which language it comes from. Low entropy neurons will have high probabilistic association to a few languages, and a very low one for most other languages.

We will rank the neurons according to their entropy taken from the distribution of probabilistic association. The neurons with the lowest entropy will be of interest.

3.2 TF-IDF

Term Frequency-Inverse Document Frequency, is a statistic commonly used in Information Retrieval to calculate the importance of a word in a document within a collection.

The TF-IDF value increases proportionally to the number of times a word appears in the document, but is counterbalanced by the frequency of the word in the corpus. This helps to find the words that are specifically important for the given document.

It consists of two parts:

Term frequency (TF) is the number of times a term t appears in a document d, normalized by the total number of terms in the document:

$$TF(t,d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

Inverse document frequency (IDF) is calculated as the logarithm of the total number of documents in the corpus divided by the number of documents containing the term:

$$IDF(t) = \log \frac{\text{Total number of documents}}{\text{Number of documents containing term } t}$$

The TF-IDF is then calculated as:

$$TF\text{-}IDF(t,d,D) = TF(t,d) \cdot IDF(t)$$

If the term t appears in many documents, it isn't a useful discriminator. Common words like articles and prepositions tend to have very low TF-IDF scores. So $IDF(t)$ will be low and consequently the $TF - IDF$ will also be low. But if a word appears in very few documents, that word is very informative. Those few documents would be useful for someone searching for that word.

If we were doing Information Retrieval, we would have a query as a list of terms, and then for every document we would add the $TF - IDF$ scores of those terms; higher values will give us more relevant documents.

We've reapplied this concept to serve our purposes as an alternative to the probabilistic approach. Instead of having a list of terms as a search query, we have a list of the top k tokens of a neuron. Instead of documents we use languages.

This way we have:

$$TF(t,l) = \frac{\text{Number of times token } t \text{ appears in language } l}{\text{Total number of terms in dataset for language } l}$$

and

$$IDF(t) = \log \frac{\text{Total number of languages}}{\text{Number of languages containing token } t}$$

This makes it so that tokens that appear in many languages are given less importance than tokens that appear in a few languages. For example, a number or symbol has low IDF, but a character in a low-resource alphabet would have high IDF.

4 Experimental Framework

4.1 Models Used

We are analyzing the 600M parameter multilanguage translation NLLB-200[23] model from Meta AI.

To perform the tokenization we also use the matching NLLB-200 tokenization model. It's important that the tokenized ID values for our data match the ones contained in the model. The model Multilayer Perceptron layers use 4096 neurons, which in the case of the top layer were the object of our analysis.

There is also a language prediction model provided by NLLB. We attempted to use it to perform the matching of tokens to languages, but its performance was very poor for single tokens.

The translation and tokenization models were obtained from HuggingFace.

4.2 Data Collection and Processing

We use a similar dataset to the one used by NLLB-200[23]. It includes parallel sentences for 108 languages, which is skewed towards English. We tokenized the entire dataset with the matching tokenization model and calculated the token counts for every language. In total we had around 243 million tokens across all languages.

The language tag tokens that are used at the beginning of each translation sentence were ignored.

5 Results

5.1 Computing Probabilities

For our probabilistic approach we must find a way of computing the probability $P(l|t)$ of observing language l given token t from our language data.

We started by observing that computing $P(t|l)$ is simpler and calculated using Bayes' rule:

$$P(l|t) = \frac{P(t|l)P(l)}{P(t)}$$

We have $P(t|l)$ the frequency of token t appearing in the corpus of language l, $P(l)$ is the proportion of tokens from language l in the entire dataset, and $P(t)$ is the frequency of token t in the entire dataset.

We found that almost all neurons were most associated with the English language, with also high associations with Turkish and Russian. The basic Bayesian approach favors languages that appear more in our dataset. About 31% of our tokens are from English data, with 24% and 15% for Turkish and Russian respectively. For comparison, Spanish tokens account for 0.7% of the data.

Using the Bayes rule as-is, the proportion of the language in our dataset is the prior probability $P(l)$. A higher prior will produce a higher posterior. This makes sense if we wanted to predict the language of tokens sampled from our dataset, but for our purposes we want the relative size of the training data to have no effect.

We solved this by taking a uniform prior:

$$\hat{P}(l|t) = \frac{P(t|l)}{P(t)\dot{N}_l}$$

With N_l being the number of languages in our data (108). The resulting $\hat{P}$ values were normalized to sum to 1 to generate the probability distribution.

5.2 Findings

We found many examples of neurons with linguistic structure, which was often easy to interpret. We will show some examples that prove the initial hypothesis that the last layer contains linguistic structure.

Our lowest entropy neuron (Figure 8) has a clear structure favoring languages of the Bantu language family. These languages are spoken by the Bantu peoples of Central, Southern, Eastern and Southeast Africa.

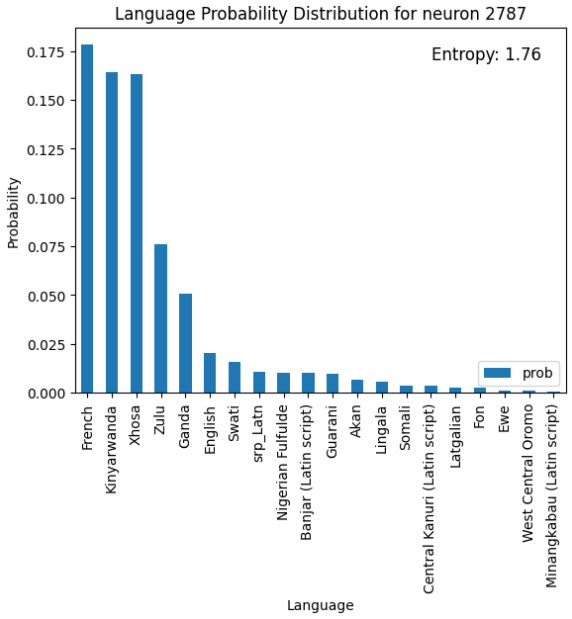

Figure 8: Probabilities for neuron grouping Bantu languages.

We can compare the results using the probabilistic metric and using tfidf. Our tfidf plot (Figure 9) shows a sharper separation between the group of significant languages and non significant languages. French is given substantially lower importance, but it still appears slightly above the Lingala language from the Banja family. Some noise is expected.

We found a clearer example of how tf-idf is superior in neuron 2949 (Figure 10 and Figure 11). This cluster has low entropy on the negative weights, meaning that

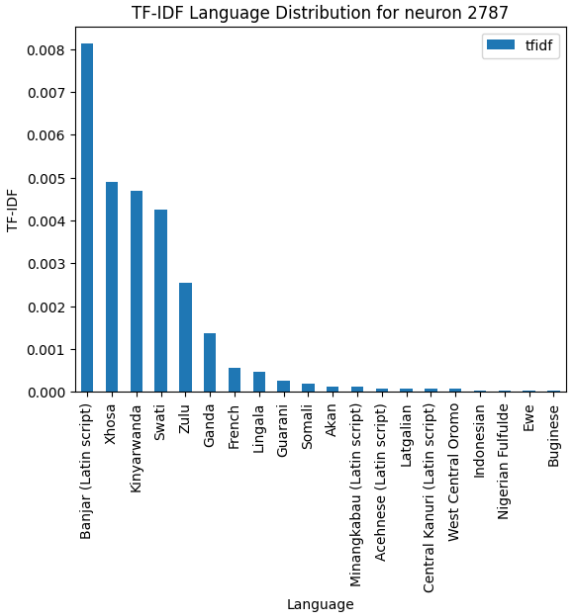

Figure 9: TF-IDF scores for neuron grouping Bantu languages.

it pulls the predicted token away and not towards the languages shown.

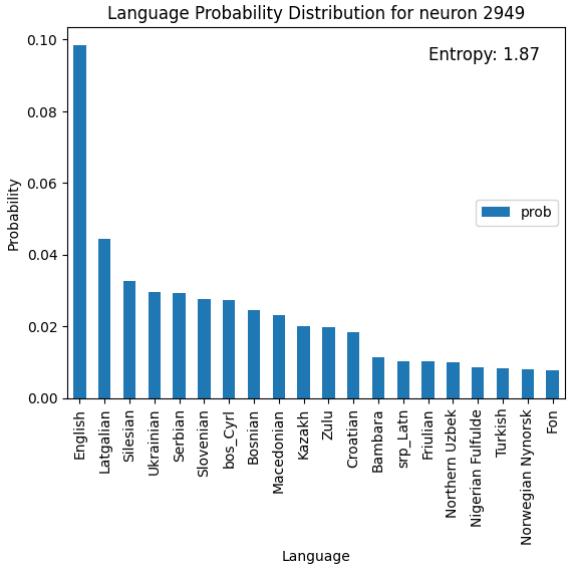

Figure 10: Probabilities for neuron Cyrillic Serbian and Bosnian.

By looking at the probabilistic distribution in Figure 10, we would think it's a neuron that tends towards English and Bosnian, two relatively unrelated languages.

The tdidf metrics show a clearer picture (Figure 11). Our neuron appears very

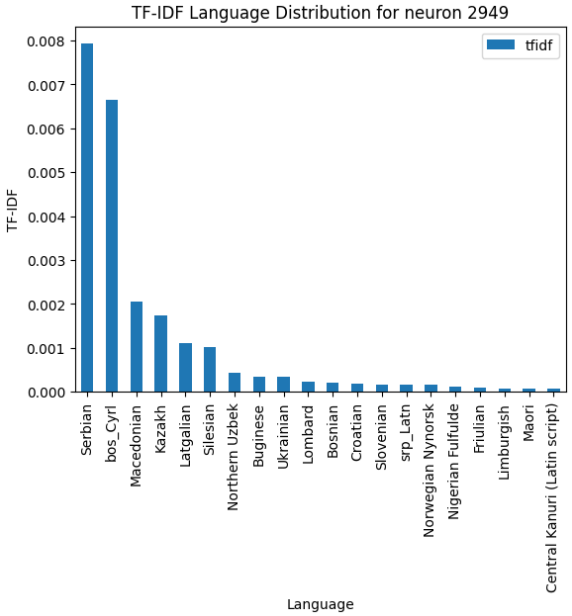

Figure 11: TF-IDF scores for neuron grouping Cyrillic Serbian and Bosnian.

correlated with Serbian and Bosnian. These two languages are so similar as to often be considered dialects of a single Bosnian-Croatian-Montenegrin-Serbian language (BCMS) [7]. The third language, Macedonian, is part of the same South-Slavic family and uses the Cyrilic alphabet as the others.

A quite interesting grouping was found in neuron 2088 (Figure 12 and Figure 13):

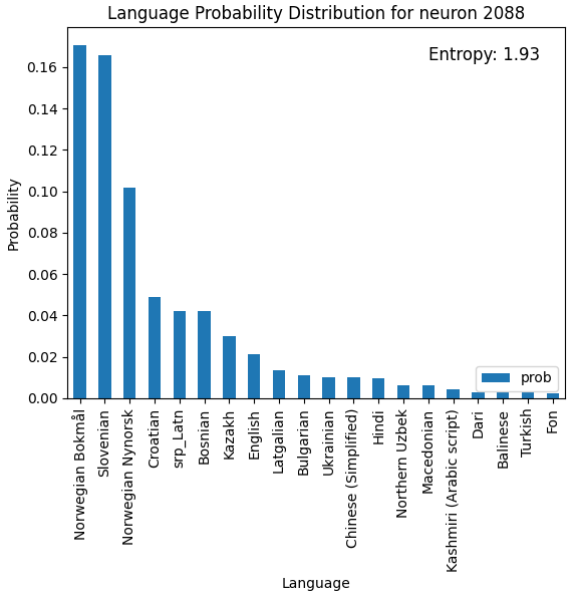

Figure 12: probability scores for neuron Norwegian and Balkan languages.

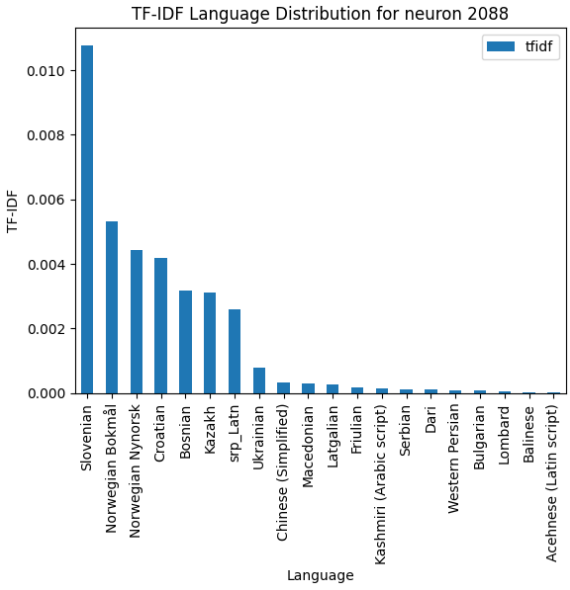

Figure 13: TF-IDF scores for neuron Norwegian and Balkan languages.

This time the difference between the probability measure and the tf-idf measure is not as significant. Both cases show that this neuron has two significant groups:

On the one hand both styles of Norwegian. The second group consists of Slovenian, Croatian, Serbian (Latin script) and Bosnian. Just as before, these are languages from the Balkan, with Croatian, Serbian and Bosnian part of the Bosnian-Croatian-Montenegrin-Serbian language (BCMS) [7].

Since we had success identifying patterns in clusters with low entropy, we now investigate ones with very high entropy. This way we find neuron 2865 with a very uniform distribution:

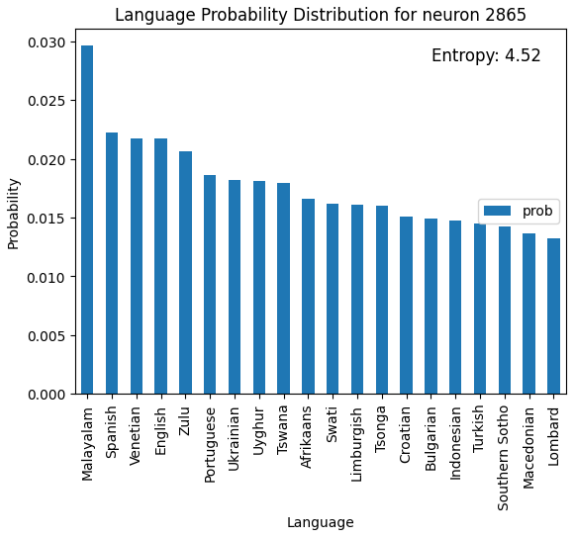

Figure 14: Probabilities for high-entropy neuron.

Neuron 2865 with entropy 4.52 looks very uniform, and we might think that it has tokens from random languages mixed together. But upon looking at the actual top tokens we find that is not the case. In fact, we find that the neuron is pushing away symbols (Table 1). It's not associated with any specific language, but we can still find a clear interpretation.

Table 1: Top tokens for neuron 2865 with very high entropy.

Token	Weight
-	-11.614623
?	-11.152328
,	-9.969774
_-	-9.656934
.	-9.521572
!	-9.239340
)	-9.115277
_/	-8.573680
”	-8.551210
:	-8.412151
...	-8.117491
_”	-8.063087
/	-7.943243
_?	-7.938666
_(	-7.788062
;	-7.735497
?”	-7.695862
’	-7.401707
_e	-7.326250

6 Conclusion and Future Work

Our analysis of individual neurons within the final feed-forward network (FFN) layer of a multilingual transformer-based Neural Machine Translation (NMT) model has provided some interesting insights into the workings of these systems. By examining neuron weights and using visualization techniques, we were able to observe patterns in how the model handles translation across multiple languages.

One of the main findings of our study is the identification of neurons that appear to influence output language choice. We found that certain neurons show stronger associations with specific output languages, suggesting a degree of specialization within the network. However, it's important to note that the relationship between individual neurons and linguistic functions is complex and not always clear-cut.

An unexpected observation was the model's apparent clustering of similar languages, as seen in the case of Serbo-Croatian variants. Despite being labeled separately in the training data, the model seemed to group these closely related languages together. This could indicate that the model is capturing some underlying linguistic similarities, though further research would be needed to confirm this and understand its implications.

This research represents a step towards better understanding multilingual NMT models, but many questions remain unanswered.

Future work could explore how these findings might be applied to improve model performance or efficiency, particularly for low-resource languages. Additionally, investigating how these patterns hold across different model architectures and training regimes could provide valuable insights.

We should consider boosting the neurons associated with low-resource languages in order to reduce hallucinations and improve performance.

Another way forward could be fine-tuning the model using Low-Resource Allocation (LoRA) [16]. This would create a language-specific matrix that gets added to our layer. We hypothesise that such LoRA weight matrix would give us further insights in the same vein as this work.

References

[1] Ba, J.L., Kiros, J.R., Hinton, G.E.: Layer normalization. arXiv preprint arXiv:1607.06450 (2016)

[2] Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 (2014)

[3] Bengio, Y.: Learning deep architectures for ai. Foundations and trends in Machine Learning **2**(1), 1–127 (2009)

[4] Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C.: A neural probabilistic language model. Journal of Machine Learning Research **3**(Feb), 1137–1155 (2003)

[5] Bishop, C.M.: Pattern recognition and machine learning. Springer (2006)

[6] Bridle, J.S.: Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In: Neurocomputing. pp. 227–236. Springer (1990)

[7] Browne, W.: Bosnian-croatian-montenegrin-serbian language. Encyclopedia Britannica, `https://www.britannica.com/topic/Bosnian-Croatian-Montenegrin-Serbian-language`

[8] Cover, T.M., Thomas, J.A.: Elements of information theory. John Wiley & Sons (2006)

[9] Elhage, N., Nanda, N., Olsson, C., Henighan, T., Joseph, N., Mann, B., Askell, A., Bai, Y., Chen, A., Drain, T., et al.: A mathematical framework for transformer circuits. Transformer Circuits Thread (2021)

[10] Ferrando, J., Sarti, G., Bisazza, A., Costa-jussà, M.: A primer on the inner workings of transformer-based language models. Workingpaper (Apr 2024). https://doi.org/10.48550/arXiv.2405.00208

[11] Gage, P.: A new algorithm for data compression. C Users Journal **12**(2), 23–38 (1994)

[12] Geva, M., Schuster, R., Berant, J., Levy, O.: Transformer feed-forward layers are key-value memories. arXiv preprint arXiv:2012.14913 (2021)

[13] Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: Proceedings of the fourteenth international conference on artificial intelligence and statistics. pp. 315–323. JMLR Workshop and Conference Proceedings (2011)

[14] Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT press (2016)

[15] Haykin, S.: Neural networks and learning machines. Pearson Upper Saddle River, 3 edn. (2009)

[16] Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Chen, W.: Lora: Low-rank adaptation of large language models. CoRR **abs/2106.09685** (2021), `https://arxiv.org/abs/2106.09685`

[17] Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)

[18] Kudo, T.: Subword regularization: Improving neural network translation models with multiple subword candidates. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 66–75 (2018)

[19] LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)

[20] Maas, A.L., Hannun, A.Y., Ng, A.Y.: Rectifier nonlinearities improve neural network acoustic models. In: Proc. icml. vol. 30, p. 3 (2013)

[21] Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems. vol. 26 (2013)

[22] Mikolov, T., Yih, W.t., Zweig, G.: Linguistic regularities in continuous space word representations. In: Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. pp. 746–751 (2013)

[23] NLLB Team, Costa-jussà, M.R., Cross, J., Çelebi, O., Elbayad, M., Heafield, K., Heffernan, K., Kalbassi, E., Lam, J., Licht, D., Maillard, J., Sun, A., Wang, S., Wenzek, G., Youngblood, A., Akula, B., Barrault, L., Mejia-Gonzalez, G., Hansanti, P., Hoffman, J., Jarrett, S., Sadagopan, K.R., Rowe, D., Spruit, S., Tran, C., Andrews, P., Ayan, N.F., Bhosale, S., Edunov, S., Fan, A., Gao, C., Goswami, V., Guzmán, F., Koehn, P., Mourachko, A., Ropers, C., Saleem, S., Schwenk, H., Wang, J.: No language left behind: Scaling human-centered machine translation (2022)

[24] Nwankpa, C., Ijomah, W., Gachagan, A., Marshall, S.: Activation functions: Comparison of trends in practice and research for deep learning. arXiv preprint arXiv:1811.03378 (2018)

[25] Peters, M.E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., Zettlemoyer, L.: Deep contextualized word representations. In: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers). pp. 2227–2237 (2018)

[26] Rosenblatt, F.: The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review **65**(6), 386 (1958)

[27] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088), 533–536 (1986)

[28] Rush, A.: The annotated transformer. Proceedings of workshop for NLP open source software (NLP-OSS) pp. 52–60 (2018)

[29] Sennrich, R., Haddow, B., Birch, A.: Neural machine translation of rare words with subword units. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 1715–1725 (2016)

[30] Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in neural information processing systems. pp. 3104–3112 (2014)

[31] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems. vol. 30 (2017)

[32] Wu, Y., Schuster, M., Chen, Z., Le, Q.V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al.: Google's neural machine translation system: Bridging the gap between human and machine translation (2016)