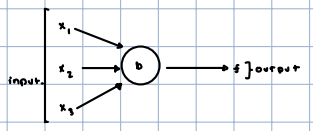
Neural Networks

Perceptron -> takes several inputs and gives a single binary output (decision)



w_j = weights

b = bias (threshold)

Basic components of a Perceptron

- input features
- weights
- summation function
- activation function
- output
- bias
- learning algorithm

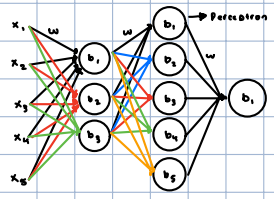
$$f = \begin{cases} 0 \text{ if } \sum_j w_j x_j \leq -b \\ 1 \text{ if } \sum_j w_j x_j > -b \end{cases} \qquad f: \begin{cases} 0 \text{ if } w \cdot x + b \leq 0 \\ 1 \text{ if } w \cdot x + b > 0 \end{cases}$$

let $z = w \cdot x + b$

Activation: $a = f(z) = \begin{cases} 0 \text{ if } z \leq 0 \\ 1 \text{ if } z > 0 \end{cases}$

- a single perceptron is a linear classifier

- a network of perceptrons is a neural network



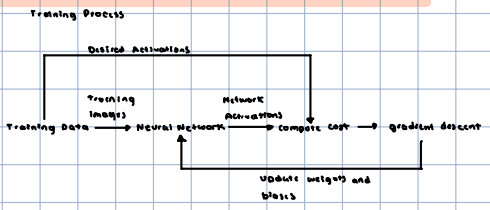
* this is called a multi-perceptron network (MLP) or a neural network

Gradient Descent

How do you optimize your weights and biases to minimize cost?

(this example assumes a single weight and bias)

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_1} & \frac{\partial C}{\partial b_1} \end{bmatrix} \qquad \Delta v = \begin{bmatrix} \Delta w_1 \\ \Delta b_1 \end{bmatrix}$$



$\Delta C = \nabla C \cdot \Delta v$

∇C = this represents the gradient of the cost function (a vector of partial derivatives) with respect to the model weights and biases

The gradient points in the direction of steepest increase of the cost function

Δv = this is the change vector containing the adjustments to be made to the weights and biases

let $\Delta v = -\eta \nabla C$

η = the learning rate determines the size of the steps taken in the direction opposite to the gradient (hence why it is negative). It is a scaler that scales the gradient vector

$\Delta C = -\eta \|\nabla C\|^2$

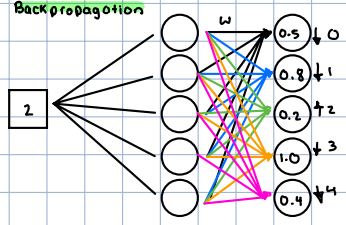
For each step:

$$w_j := w_j - \eta \frac{\partial C}{\partial w_j}$$

$$b_j := b_j - \eta \frac{\partial C}{\partial b_j}$$

* gradient descent is very computationally heavy and can make training a model very difficult
  - this because you have to take the derivative of the cost with respect to each weight and bias

Backpropagation



Backpropagation - is the algorithm is how a single training example would like to nudge the weights and biases

- we want to increase the activation of 0.2 which is a weighted sum

$0.2 = w_0 + w_1 a_1 + w_2 a_2 + w_3 a_3 + w_4 a_4 + w_5 + a_5$

↳ this then put into a sigmoid activation

- There are 3 different avenues that can be used to increase the activation value of 2:
  1. Increase the bias
  2. Increase the weights
  3. Change the activations from the previous layer

* Notice certain weights have differing levels of influence based on the activations of the previous layer

* larger activations from the previous layer have the biggest effect on the activation of the current layer

* this also means increasing the weights corresponding to larger activations will have a larger effect on the output layer

* increase w_j in proportion to a_j

* change a_j in proportion to w_j

* remember we also want all of the last layer neurons to be less active (answering not 2)
  - each of the output neurons has its own thoughts and beliefs of what should happen to that second to last layer

* the desire of the output 2 neuron is added together with all the other output neurons in proportion to the weights and in proportion to how much each of those neurons needs to change

Backpropagation - the desire of the digit 2 neuron is added together with the desire of all the other output neurons for what should happen to this second to last layer in proportion to the corresponding weights. This is what is called propagating backwards.

By adding together all of the desired effects we get a list of nudges of what you want to happen to this second to last layer. Once you have those you can recursively apply the relevant weights and biases that determine those values

$$-\eta \nabla C(w_1, w_2, \ldots w_{13,000}) = \begin{bmatrix} -0.08 \\ +0.12 \\ -0.05 \\ \vdots \\ +0.04 \end{bmatrix}$$

Average over all training data

* Remember this is for a single training example

You are supposed to do this backprop routine for every training example recording how they would like to change weights and biases and then you average together those changes to the weights. The collection of the average nudges to each weight and bias is loosely speaking the negative gradient of the cost function

Stochastic Gradient Descent -> breaking up your training data into small batches and then computing the gradient (won't give you the exact gradient of the cost function) but it is a good approximation

* for backpropagation to work you need a lot of training data

Transformer Neural Networks

- transformer is a type of neural network, and neural networks only input numbers for input values

1. Concept of Word Embeddings

Word embeddings are a form of representation that allows words with similar meaning to have a similar representation. They are essentially vectors of real numbers where each word in the language is mapped to a vector in a predefined vector space. Each vector captures the semantic properties of words that are semantically similar in a pre-defined vector space. Each vector captures the semantic properties of words: words that are semantically similar are placed closer together in the space

2. Tokenization

Before a text can be processed by a transformer, it needs to be tokenized. Before text can be fed into a transformer it must be converted from raw strings of characters into tokens. This process is known as tokenization

- word level tokenization
- subword tokenization (very commonly used)
- character level tokenization

Once a text is tokenized, each token is converted into a numerical form known as a token ID. The mapping from tokens to token IDs is defined by the vocabulary that the model was trained on. The vocabulary contains a unique ID for each token.

- a vocabulary is a list of all unique tokens that the model will recognize. Each token in this list is assigned a unique index or ID. The vocabulary is typically created during the training of the model on large corpus of text data, where the tokenization method is applied to the text, and frequency statistics are often used to decide which tokens are included in the vocabulary
  - Vocabulary size: the size of the vocabulary is a critical parameter. A large vocabulary can capture more information but requires more computational resources; a smaller vocabulary is more computationally efficient but may lose information if many words are mapped to an "unknown" token

Mapping tokens to token IDs

- with a vocabulary established, each token in a text input is replaced by its corresponding ID based on the vocabulary list. This process is straightforward: the tokenizer scans through the input text, segments it and then substitutes each token with its index in the vocabulary. If a token does not exist in the vocabulary, it is replaced by the ID for a special token

3. Token IDs to high dimensional vectors (embedding layer)

- Once each piece of text has been converted to token IDs, these IDs are then input into the embedding layer of the transformer model. This embedding layer contains a lookup table where each unique ID corresponds to a vector in high dimensional space
- These vectors are not merely arbitrary; they are learned and contain semantic meaning. This transformation allows the model to process text in a meaningful way
- If the transformer is trained from scratch, these vectors are initially random, but are adjusted during training to capture useful properties and relationships

4. Positional Encoding

- Once each token is represented by its corresponding vector from the embedding layer, positional encodings are added to these vectors. Positional encodings provide the model with information about the order or position of tokens within the sequence. This is crucial because the self-attention mechanism in transformers does not process sequential data in order (unlike RNNs)
- we add a set of numbers that correspond to word order to the embedding values for each word
- the transformer model uses sinusoidal functions to compute positional encodings, which are then added to the embeddings of the tokens. Specifically, for each dimension of the positional encoding vector:
  - For even indices $2i$:

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

  - For odd indices $2i+1$:

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

what each variable means:

- pos is the position of the token (vector w semantic meaning) in the sequence
- i is the index within the positional encoding vector
- d_{model} is the dimension of the embedding

the dot product of a vector can be used to define how well two tokens align with each other

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} v_1 \cdot w_1 \\ + \\ v_2 \cdot w_2 \\ + \\ \vdots \\ v_n \cdot w_n \end{bmatrix} = \text{a single value}$$

- if this value is positive if vectors point in similar directions
- if the value is 0 then the vectors are perpendicular
- if the dot product is 0 then the vectors point in different directions

- a network can only process a fixed number of vectors at a time known as its context size
  - in gpt-3 the context size was 2,048 (number of tokens being processed at once)

- By adding positional encodings to the embedding of tokens, the transformer model can understand not only what each token represents (through its embedding) but also where it fits in the overall sequence (through its positional encoding). This is crucial for effectively handling tasks that depend on the sequence order, such as translating sentences or answering questions based on a given text

5. Self-attention - what you need to add to your generic embeddings so it actually represents the true proper meaning of the word

- what is self attention (high level overview)
  - Self attention sometimes referred to as intra-attention, is a mechanism that allows each token in the input sentence to interact w every other token to determine its influence on the sequence. This process helps the model to dynamically focus on different parts of the sequence as needed without being restricted by their position in the sequence
- How self attention is connected to the previous steps
  - the input to the self-attention layer consists of the output from the embedding and positional encoding steps. Each token representation now contains information about the tokens meaning (from embeddings) and its position in the sequence (from positional encodings)
  - Matrix Representation: In practice, self-attention is computed using matrices. The input matrix X represents all embedded tokens in the sequence, adjusted for their positions. For each token, three different vectors are derived through linear transformations: Query (Q), Key (K), and Value (V). These transformations are performed using trainable weights, typically different for each attention head in multi-head

attention setups.

- In the vector X
  - each row in X corresponds to the embedded vector of a token. The first row of X contains the embedding (plus positional encoding) of the first token, the second row contains that of the second token, and so on

Detailed computation steps

1. Each token's embedded vector is transformed into 3 different vectors: Query (Q), Key (K), and value (V). These values are used to determine the attention scores
   - Q (Query) $= X W^Q$
   - $K = X W^K$
   - $V = X W^V$

Here W^Q, W^K, and W^V are parameter matrices that are learned during training

Why break up into 3 different vectors

- In the self-attention mechanism of a transformer, each input vector is transformed into the 3 vectors above, below is what they signify:
  1. Query (Q): a query is a representation of a token used to score how well it matches every other token. Essentially when considering a specific token; it query vector is used to seek out which tokens (keys) are most relevant to it
  2. Key (K): A key is associated with a token to be matched against queries. When a query is compared to this key, the resulting score determines the impact of the corresponding token's value on the output
  3. Values (V): A value is a representation of a token that is used to compute the final output. Once tokens are scored based on the query-key matches, the values are weighted by these scores and summed up to produce the output for the next layer

Attention scores determine how much focus to place on other parts of the input sequence when processing a specific part of that sequence.

- attention score calculation $= \frac{Q K^T}{\sqrt{d_k}}$

the scores are scaled down by the square root of the dimension of the key vectors (d_k) to stabalize the gradients during training

Forward Pass

- For each input token, the model calculates the query, key and value vectors using the current values of W^Q, W^K, and W^V
- Self-attention scores are computed based on the dot product of queries and keys, scaled by the square root of the dimension of the key vector
- these scores are normalized using a softmax function to create a distribution of weights

Example

- a fluffy blue creature roamed the verdant forest

E1 E2 E3 E4 E5 E6 E7 E8 — E_i → word embedding + positional encoding for word i

$\vec{E_1}$ $\vec{E_2}$ $\vec{E_3}$ $\vec{E_4}$ $\vec{E_5}$ $\vec{E_6}$ $\vec{E_7}$ $\vec{E_8}$

the goal is to create a new set of embeddings where each word considers the words around it to determine its meaning (Query vectors)

to compute this take a weight vector W_Q and multiplying it by the embedding vectors

the entries of this matrix are parameters of the model (meaning true behavior is learned from data)

$W_Q \cdot E_1 = \vec{Q_1}$ ← query vector

do this computation until you get a query vector for every token embedding

$[1 \times 512] \times [512 \times 64] = [1 \times 64]$

- a fluffy blue creature roamed the verdant forest

E1 E2 E3 E4 E5 E6 E7 E8 — W_Q → $\vec{Q_1}$ $\vec{Q_2}$ $\vec{Q_3}$ $\vec{Q_4}$ $\vec{Q_5}$ $\vec{Q_6}$ $\vec{Q_7}$ $\vec{Q_8}$

→ conceptually this is checking if there are any words that change the words meaning

- W_K = key matrix

a → $\vec{E_1} \cdot W_K = K_1$ — $K_1 \cdot Q_1$ $K_1 \cdot Q_2$ $K_1 \cdot Q_3$ $K_1 \cdot Q_4$ $K_1 \cdot Q_5$ $K_1 \cdot Q_6$ $K_1 \cdot Q_7$ $K_1 \cdot Q_8$

fluffy → $\vec{E_2} \cdot W_K = K_2$ — $K_2 \cdot Q_1$ $K_2 \cdot Q_2$ $K_2 \cdot Q_3$ $K_2 \cdot Q_4$ $K_2 \cdot Q_5$ $K_2 \cdot Q_6$ $K_2 \cdot Q_7$ $K_2 \cdot Q_8$

blue → $\vec{E_3} \cdot W_K = K_3$ — $K_3 \cdot Q_1$ $K_3 \cdot Q_2$ $K_3 \cdot Q_3$ $K_3 \cdot Q_4$ $K_3 \cdot Q_5$ $K_3 \cdot Q_6$ $K_3 \cdot Q_7$ $K_3 \cdot Q_8$

creature → $\vec{E_4} \cdot W_K = K_4$ — $K_4 \cdot Q_1$ $K_4 \cdot Q_2$ $K_4 \cdot Q_3$ $K_4 \cdot Q_4$ $K_4 \cdot Q_5$ $K_4 \cdot Q_6$ $K_4 \cdot Q_7$ $K_4 \cdot Q_8$

roamed → $\vec{E_5} \cdot W_K = K_5$ — $K_5 \cdot Q_1$ $K_5 \cdot Q_2$ $K_5 \cdot Q_3$ $K_5 \cdot Q_4$ $K_5 \cdot Q_5$ $K_5 \cdot Q_6$ $K_5 \cdot Q_7$ $K_5 \cdot Q_8$

⋮

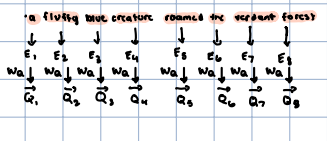
all these dot products are being computed parallel

= dot product values highlighted with reds are high values

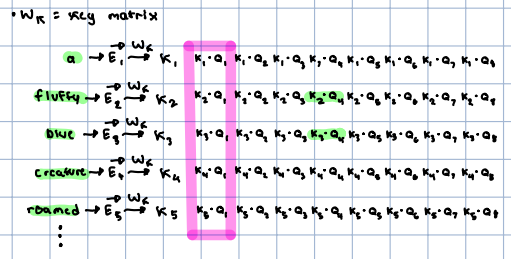
- conceptually task is answering to the query whether or not a word is changing its meaning

* If the dot product of a key and query vector is really high then the specific key embeddings attend to the query embeddings

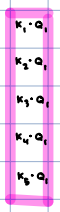
- As of right now the grid of values ranges from $-\infty$ to ∞ giving us a score for how relevant each word is to updating the meaning of every other word
- Next we will take a weighted sum along each column, weighted by relevance (so we want each column to add up to one, as if they were a probability distribution)



- W_K = key matrix



normalized columns called an attention pattern



apply a softmax function to each column

- to normalize each column apply a softmax function to each column so it adds up to one
- at this point its safe to think about each column as giving weights according to how relevant the word on the left is to the corresponding value at the top

Softmax function - how it works

- what is the goal of a softmax function?
  - softmax is a way to change a valid list of numbers into a valid probability distribution such that the largest values end up having a higher probability (closer to 1) while, the smallest values have a smaller probability (closer to 0)
- take your output vector (doesn't add up to 1) ex. $K_1 \cdot Q_1 = x_1$ and raise e to the power of each of the values in the vector which makes all values positive:

$$e^{x_1} > 0$$

then take the sum of all e^{x_n} values

$$\sum_{n=0}^{n-1} e^{x_n}$$

and divide each e^{x_n} value by the sum

$$e^{x_n} / \sum_{n=0}^{n-1} e^{x_n}$$

which normalizes the vector which adds up to 1

- In the paper "Attention is all you need" all of the above visuals are represented in the following equation:

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

- Q represents a full matrix of query vector embeddings
- K represents a full matrix of key vector embeddings
  - QK^T → represents the dot product between the key and query matrices
  - $\sqrt{d_K}$ → divide QK^T by $\sqrt{d_K}$ for numerical stability
    - d_K = dimension in the key vector space
  - then the softmax that's wrapped around QK^T normalizes the dot product between the key and query matrices

Softmax With Temperature

- The temperature T modifies the softmax function by dividing the logits by the temperature value before applying the softmax:

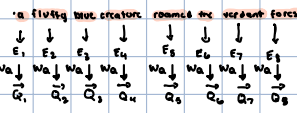
$$\sigma(z_i, T) = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

- The temperature plays a role in the "sharpness" of the probability distribution:
  - low temperature ($T < 1$): the softmax output becomes more confident, with larger differences between the maximum probability and other probabilities. Lower probabilities make the output distribution "sharper" or more peaked, which increases the model's confidence in its predictions
  - high temperature ($T > 1$): the softmax output becomes smoother and more uniform. Higher temperatures make the models output probabilities closer to each other, which can be useful when you want the model to be less confident and explore more diverse options

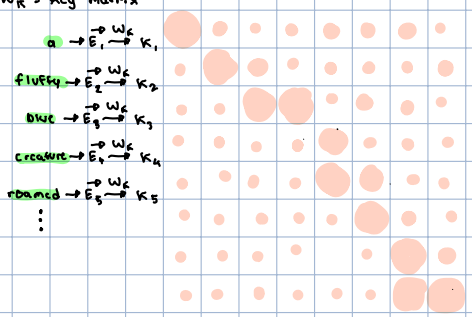
- During the training process when you run this model on a given text example and weights are being tuned:
  - it makes easier to train the model with a sentence from the ground up
  - example: if you have the sentence - "a fluffy blue creature roamed the verdant forest"
    - the fluffy → 999
    - the fluffy blue → 999
    - the fluffy blue creature → 999
    - the fluffy blue creature roamed → 999
    - the fluffy blue creature roamed the → 999
    - the fluffy blue creature roamed the verdant → 999
    - the fluffy blue creature roamed the verdant forest → 999

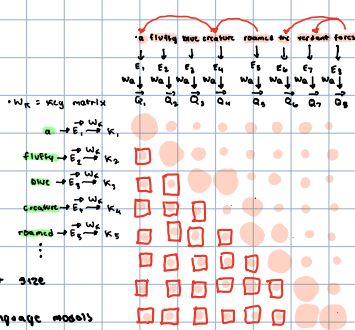
* training process becomes a lot more efficient if you simultaneously have the model predict every possible next token following each initial subsequence of tokens in this passage
* what would effectively act as a single training example would now act as many training examples

- the bigger the circle the larger the dot product



- W_K = key matrix





* the size of the attention grid is equal to the square of the context size
  - this is why context size can be a huge bottleneck for large language models and scaling it up is a non-trivial task

Masking

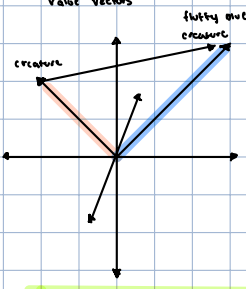
- GPT models are autoregressive language models. This means that the prediction of a word at a specific position should only depend on the words before it and not on any future words.
- Masking is used within the self-attention layers to prevent positions from attending to subsequent positions. This means that for a given word, the attention mechanism can only consider previous words in the sequence
  - How masking is created: A mask is created where the mask at position i,j in the matrix is set to 0 if $i \geq j$ and $-\infty$ if $i < j$. The mask is applied before the softmax step in the attention mechanism
- During the softmax calculation of the attention scores, the $-\infty$ will effectively become zero after applying the softmax function, which means that these positions will not contribute to the weighted sum in the attention output
- this is essential because w/o it the model might learn to rely on future inputs that will not be available during text generation, leading to poor performance and non-sensical outputs

- if you never want later words to influence earlier words, otherwise they could give away the answer for what comes next.

♥

- that means the spots in ☐ represent later tokens intervening earlier ones, should be forced to be 0
- its not possible to force the red boxes to be equal to 0 because then the normalized columns will not add up to 1 anymore
- the most common way to fix this issue is before applying the softmax function, you set all the entries in the ☐ to be negative infinity $-\infty$, now after applying the softmax function, all the values in the ☐ that are set to $-\infty$ will get turned to 0, while keeping the column normalized

Value vectors



take your W_V (value weight vector) and multiply it by your word embedding of creature

$$W_V \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14}\cdots x_{1i} \\ x_{21} & x_{22} & x_{23} & x_{24}\cdots x_{2i} \\ x_{31} & x_{32} & x_{33} & x_{34}\cdots x_{3i} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_i \end{bmatrix}$$

Value Vector

*if this word is relevant to adjusting the meaning of something else, what exactly should be added to the embedding of that something else in order to reflect this

- a fluffy blue creature roamed the verdant forest

E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8

$W_Q \vec{E}_1 = \vec{Q}_1$ $W_Q \vec{E}_2 = \vec{Q}_2$ $W_Q \vec{E}_3 = \vec{Q}_3$ $W_Q \vec{E}_4 = \vec{Q}_4$ $W_Q \vec{E}_5 = \vec{Q}_5$ $W_Q \vec{E}_6 = \vec{Q}_6$ $W_Q \vec{E}_7 = \vec{Q}_7$ $W_Q \vec{E}_8 = \vec{Q}_8$

a $\rightarrow \vec{E}_1 \xrightarrow{W_V} V_1$

fluffy $\rightarrow \vec{E}_2 \xrightarrow{W_V} V_2$

blue $\rightarrow \vec{E}_3 \xrightarrow{W_V} V_3$

creature $\rightarrow \vec{E}_4 \xrightarrow{W_V} V_4$

roamed $\rightarrow \vec{E}_5 \xrightarrow{W_V} V_5$

$0.00 \cdot \vec{V}_1$

$0.42 \cdot \vec{V}_2$

$0.58 \cdot \vec{V}_3$

$0.00 \cdot \vec{V}_4$

$0.00 \cdot \vec{V}_5$

$= \Delta\vec{E}_4$

a change that you want to add

creature

$\vec{E}_4$ (original embedding matrix)

+

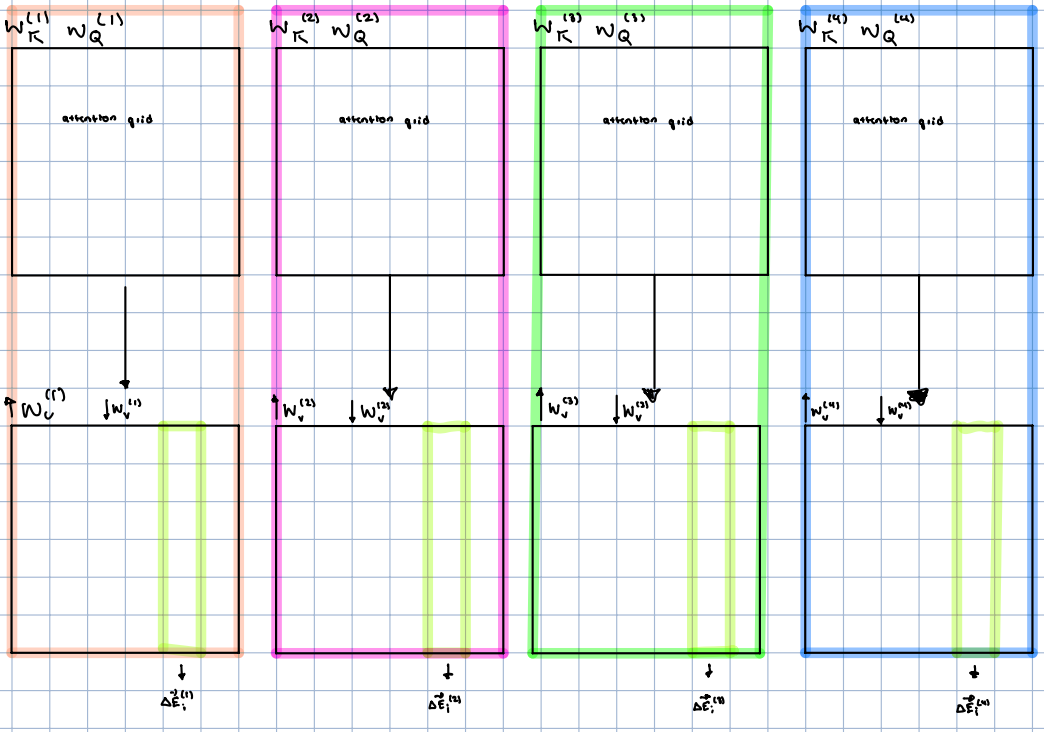
$\Delta\vec{E}_4$ (weighted sum w/ value matrix)

=

$\vec{E}_4$: this should be a more refined vector with a more contextually rich meaning

*you do this for every embedding token

this whole process is a single head of attention



*every head proposes a change to the original vector embedding that will be added to the original embedding

$$\vec{E}_i = \Delta\vec{E}_i^{(1)} + \Delta\vec{E}_i^{(2)} + \Delta\vec{E}_i^{(3)} + \Delta\vec{E}_i^{(4)} + \ldots$$

(however many attention heads)

*each color is an attention head, and by running many heads in parallel you can learn many distinct ways, your giving the model the capacity to learn many distinct ways that context changes meaning

new embedding = $\vec{E}_i + \Delta\vec{E}_i^{(1)} + \Delta\vec{E}_i^{(2)} + \Delta\vec{E}_i^{(3)} + \Delta\vec{E}_i^{(4)} + \ldots$

- this new embedding will capture the meaning of the embedding matrix

* this diagram shows the architecture of how the model is being trained

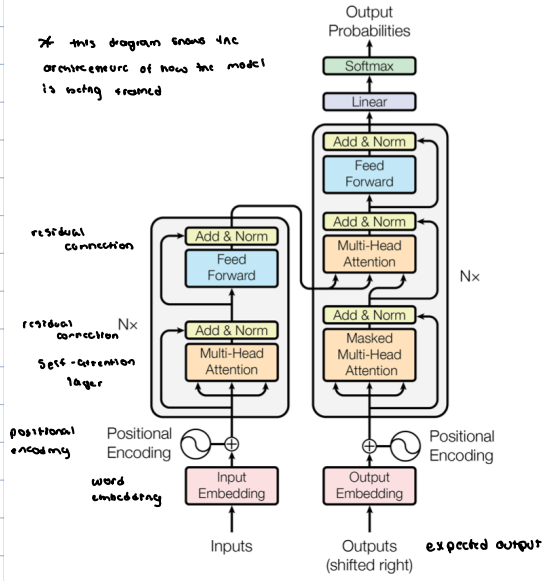
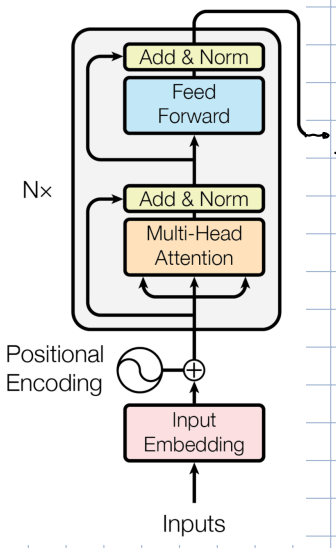


Figure 1: The Transformer - model architecture.

The 4 features labeled next to the encoder allow the transformer to: encode words into numbers, encode the position of the words, encode the relationships among the words, and relatively easily and quickly train in parallel.



this right here is called the encoder

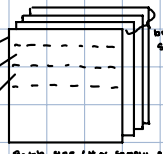
lets say we are translating a sentence from english to spanish

*English sentence: My name is Pranav → after this sentence is passed into the encoder, each word will be turned into a numerical vector that has the contextual meaning of the word, the positional encodings of the words, and the relationships with other words encoded into each token vector

*Spanish sentence: Me llamo Pranav

expected output

Word embedding tensor



Padding tokens - are used in the input of the decoder during training primarily to handle sequences of varying lengths in a batch. Processing sequences in batches is much more efficient than processing each one at a time. Padding facilitates this batching by ensuring that all data in the batch can be represented in a uniformly shaped tensor.

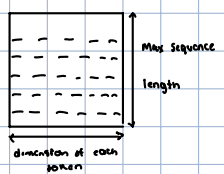
Batch size (# of samples at a time we can pass through the network at a time → all being processed in parallel)

X

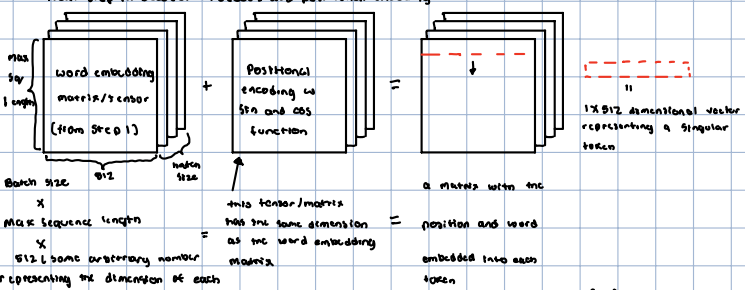
Max sequence length

X

512 (some arbitrary number representing the dimension of each vector)

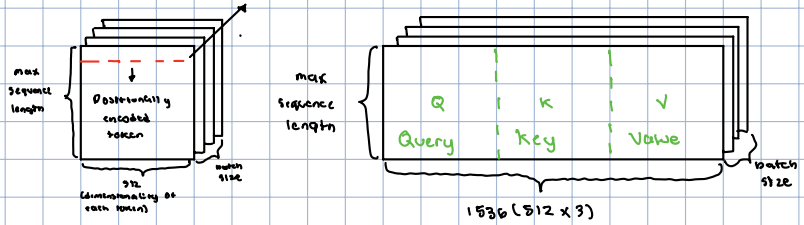


Next step in Decoder Process: add positional encoding



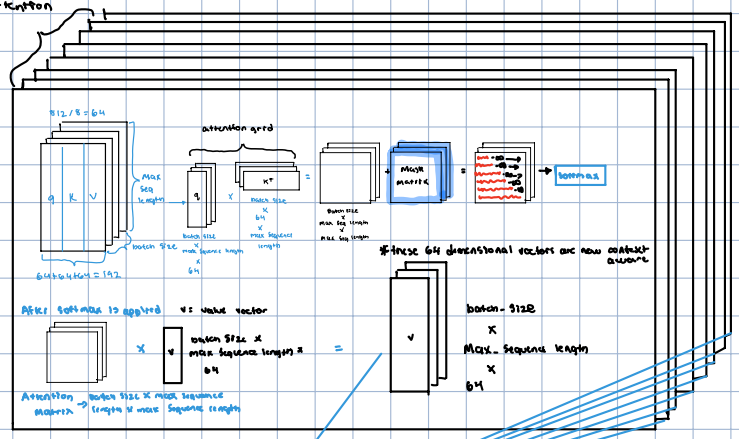
* Query, key and value vector can be intuitively interpreted as the same as before

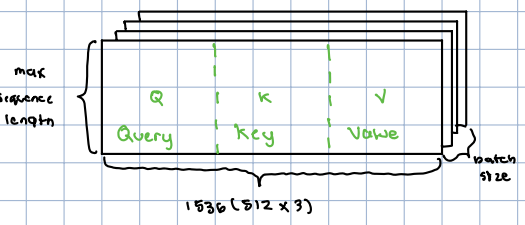
We need each of these vectors to be represented by 3 more vectors



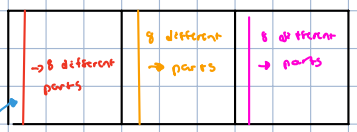
*the next step will be continued onto the next page

multi-attention block 8 heads all being run parallel



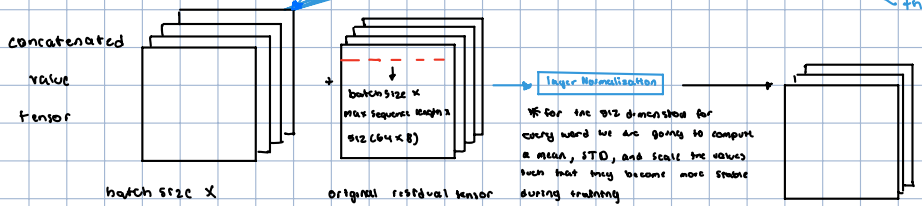


*you want to break each Query, Key, and value matrix into 8 different vectors (so that means you should divide on the vertical dimension)



→ so total this is divided into 24 different vectors

this is equal to the Query vector of the first head



batch size x max sequence length x 512 (64 x 8)

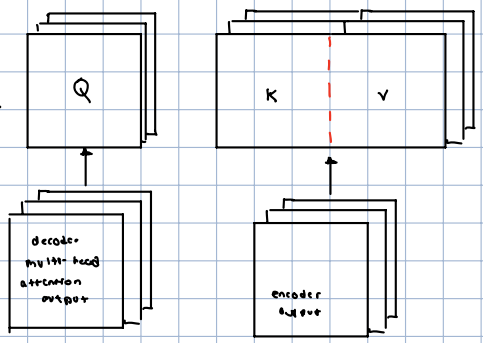
original residual tensor (matrix w each row representing a word that has gone through encoding and positional encoding)

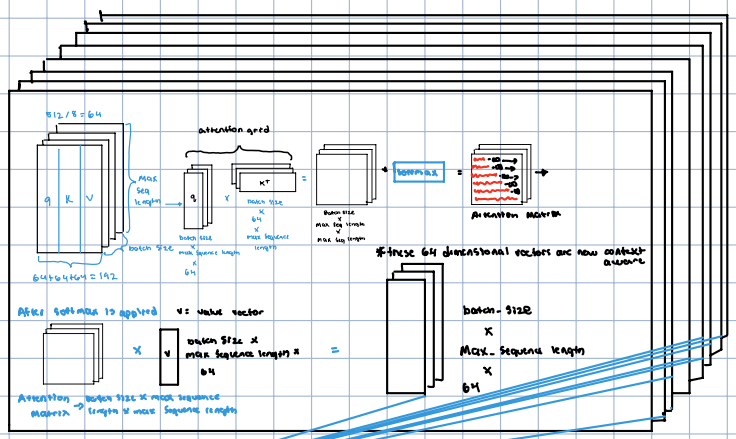
If for the 512 dimension for every word we are going to compute a mean, STD, and scale the values such that they become more stable during training

↳ each word represents 512 word vector that is extremely contextually aware

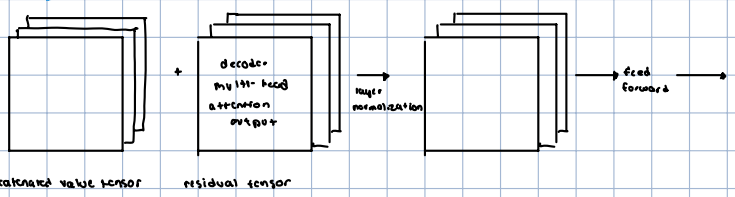
Multi-Head Cross-Attention

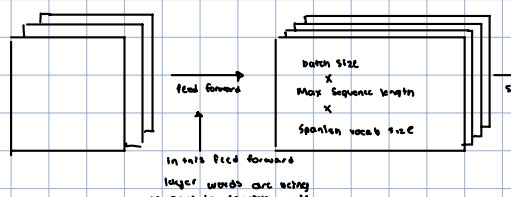
*you can use the output from your softmax function as your query vector





now a probability distribution where each row represents the next predicted word





in turn feed forward layer words are being mapped to spanish words