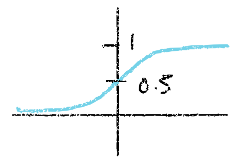
Logistic Regression

The logistic function is an S-shaped or 'sigmoid' function, which takes any real input z, and outputs a value between zero and one.

$$f(z) \text{ or } \sigma(z) = \frac{1}{1+e^{-z}} \xrightarrow{\text{graphs to}}$$



Let z be the linear combination of the input features:

$$z = \bar{w}^T \bar{x} + b = \sum_{i=1}^{n} w_i x_i + b$$

Here,

$\bar{w} = [w_1, w_2, \ldots, w_n]$ is the vector of weights

$\bar{x} = [x_1, x_2, \ldots, x_n]$ is the input feature vector

b is the bias term.

So σ squashes z into the range $(0, 1)$.

Classifying 0, or 1

$$P(y=1 \mid x) = \sigma(z) = \frac{1}{1+e^{-(\bar{w}^T \bar{x} + b)}}$$

$$P(y=0 \mid x) = 1 - \sigma(z)$$

In this way, the sigmoid function can be used to model the binary classification task, where

$$\hat{y} \begin{cases} 1 & \text{if } \sigma(z) \geq 0.5 \\ 0 & \text{if } \sigma(z) < 0.5 \end{cases}$$

The model parameters (weights $\bar{w}$ and bias b) can be trained through binary cross-entropy. (BCE)

$$L(y, \hat{y}) = -[y \log(\hat{y}) + (1-y)\log(1-\hat{y})]$$

$\hookrightarrow \sigma(z)$

Where:

- y is the true label (either 0 or 1)
- $\hat{y}$ is the predicted probability of the class being 1.

Deriving BCE:

The probability of the true class label y can be modeled as:

$$P(y \mid X, \bar{w}) = \begin{cases} \hat{y}, & \text{if } y = 1 \\ 1-\hat{y}, & \text{if } y = 0 \end{cases}$$

Which can be re-written as:

$$P(y \mid X, \bar{w}) = \hat{y}^{y}(1-\hat{y})^{1-y}$$

Assuming we have dataset with N samples, the likelihood $\mathcal{L}$ of observing the true labels $y_1, y_2, \ldots, y_N$ given the probabilities $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_N$, is the probability of the individual probabilities:

$$\mathcal{L}(\bar{w}) = \prod_{i=1}^{N} P(y_i \mid \bar{x}_i, \bar{w}) = \prod_{i=1}^{N} \hat{y}_i^{\,y_i}(1-\hat{y}_i)^{1-y_i}$$

For easier computation, we apply log

$$\log \mathcal{L}(\bar{w}) = \sum_{i=1}^{N} \log\left(\hat{y}_i^{y_i} (1-\hat{y}_i)^{1-y_i}\right)$$

$$= \sum_{i=1}^{N} \left(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right)$$

Finally, we apply a negative which makes our goal minimizing the negative log-likelihood. This makes the optimization task more suitable for gradient descent.

During training, the goal is to minimize the BCE loss by adjusting the model's parameters $\bar{w}$.

$$\bar{w} \leftarrow \bar{w} - \alpha \nabla_{\bar{w}} \mathcal{L}(\bar{w})$$

where

- $\mathcal{L}(\bar{w}) = -\frac{1}{N}\sum_{i=1}^{N} \left(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right)$
- α is the learning rate
- $\nabla_{\bar{w}}\mathcal{L}(\bar{w})$ is the gradient of the loss function with respect to $\bar{w}$.

Let $\ell(\bar{w})$ be the single sample BCE loss:

$$\ell(\bar{w}) = -\left(y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right)$$

Then, calculate the derivative of $\ell(\bar{w})$ w.r.t. $\bar{w}$:

Chain rule: $\dfrac{\partial \ell(\bar{w})}{\partial \bar{w}} = \dfrac{\partial \ell(\bar{w})}{\partial \hat{y}} \dfrac{\partial \hat{y}}{\partial z} \dfrac{\partial z}{\partial \bar{w}}$

$$\frac{\partial \ell(\bar{w})}{\partial \bar{w}} = \frac{\partial \ell(\bar{w})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial \bar{w}}$$

1 $$\frac{\partial \ell(\bar{w})}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right) = \frac{\hat{y}-y}{y(1-\hat{y})}$$

2 $$\frac{\partial \hat{y}}{\partial z} = \hat{y}(1-\hat{y})$$

3 $$\frac{\partial z}{\partial \bar{w}} = \bar{x}$$

$$\frac{\partial J(\bar{w})}{\partial \bar{w}} = \bar{x}(\hat{y}-y)$$

For the full batch:

$$\nabla_{\bar{w}} \mathcal{L}(\bar{w}) = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)\, x_i$$

This final gradient descent is actually quite intuitive, simply averaging the distance between the correct label and the predicted label.

To generalize this to multiclass classification, all we need to do is change $\hat{y}$ to the softmax function, and recalculate the gradient descent.

The softmax function is a generalization of the logistic (or sigmoid) function for multiple dimensions. In neural networks, it is used to <u>normalize</u> the output of a network to a probability distribution over predicted output classes.

meaning,

each vector component will be in the interval $(0,1)$, and will add up to 1.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

The softmax function applies the standard exponential function to each element z_i of the input vector $\bar{z}$, and normalizes these values by dividing the sum of all these exponentials. This normalization ensures that the sum of the components of the output vector $\sigma(\bar{z})$ is 1.

"Softmax" derives from the amplifying effects of the exp. on any maxima in the input vector.

For example,

$$\text{softmax}((1,2,8)) = (0.001, 0.002, 0.997)$$

We don't necessarily have to use e as base, any $b>0$ can be used. Such b can be expressed as e^{β}, for any real β. But e ensures easy derivatives, and logs.

Deriving ∇ for multiclass classification with softmax:

1. Negative log-likelihood

① $P(y \mid \bar{x}, \bar{w}) = \prod_{i=1}^{N} P(y_i \mid x_i, \bar{w}_i) = \prod_{i=1}^{N} \hat{y}_{i,k_i}$

where k_i is the index of the true class for sample i

② $\log \mathcal{L}(\bar{w}) = \sum_{i=1}^{N} \log(\hat{y}_{i,k_i}) = \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log(\hat{y}_{ik})$

where y_{ik} is 1 if true class is k for sample i, and 0 otherwise.

③ Apply the negative

$$-\mathcal{L}(\bar{w}) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log(\hat{y}_{ik})$$

2. Computing the gradient

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_{ik}} = \hat{y}_{ik} - y_{ik}, \quad \frac{\partial \hat{y}_{ik}}{\partial z_j} = \hat{y}_{ik}(\delta_{jk} - \hat{y}_{ij})$$

↳ Kronecker delta: 1 if $j=k$, 0 otherwise

$$\frac{\partial \mathcal{L}}{\partial \bar{w}} = \sum_{i=1}^{N} \sum_{k=1}^{K} (\hat{y}_{ik} - y_{ik}) \frac{\partial \hat{y}_{ik}}{\partial \bar{w}}$$

which leads to the overall gradient

$$\frac{\partial \mathcal{L}}{\partial \bar{w}} = \sum_{i=1}^{N} (\hat{y}_i - y_i) x_i$$

Neural Networks

We now have an understanding of the way logistic regression is used for binary classification. But how can this be generalized to other tasks?

Let a neuron be a thing that holds a number between 0 and 1. Let the value be the neuron's activation.

In each layer of our neural network, we have a series of neurons. The first layer of a neural network for a number classification task would be the pixel activations in the input image, and the last layer would be the activation of digits 0-10 as a series of neurons. Layers in between are hidden layers. Neurons in this layer detect higher-level patterns in the input image.

These layers are connected to each other with weights. The network connects every neuron in the previous layer to all neurons in the next. Thus, the activation of a given neuron is the weighted sum of the activations of all neurons in the previous layer. We call this structure the multilayer perceptron.

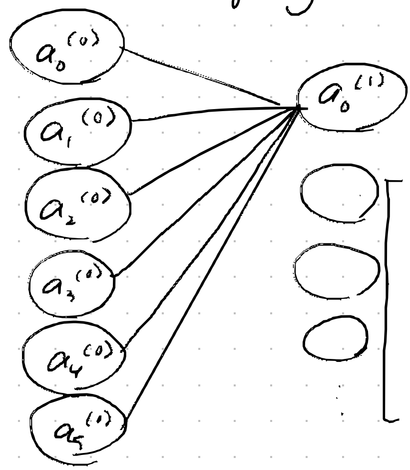
Given some activation $w_1 a_1 + w_2 a_2 + w_3 a_3 + \cdots w_n a_n$, where w_i are weights and a_i are activations of neurons from the previous layer, we put these through the sigmoid function, so that the value is squished into $(0, 1)$.

We also throw a bias term to the mix, some integer b, such that it ensures the neuron to activate meaningfully only when it reaches b. for example,

$$\sigma(w_1 a_1 + w_2 a_2 + \cdots + w_n a_n - 10)$$

only activate meaningfully when weighted sum > 10.

What this vanilla neural network allows is a scaling up of parameters for much more complex tasks. Since we have so many sigmoid functions to write at this scale, we simplify the notation.



$$a_0^{(1)} = \sigma\left(\underline{w_{0,0}\, a_0^{(0)} + w_{0,1}\, a_1^{(0)} + \cdots + w_{0,n}\, a_n^{(0)} + b_0}\right)$$

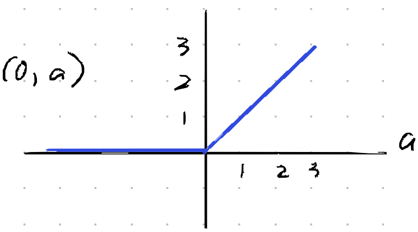
$$\underset{W}{\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix}} \underset{\bar{a}^{(0)}}{\begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix}} + \underset{\bar{b}}{\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}}$$

So the new layer's activation simplifies to:

$$\bar{a}^{(1)} = \sigma(W\bar{a}^{(0)} + \bar{b})$$

A modern alternative of the sigmoid is the ReLU function.

$$\text{ReLU}(a) = \max(0, a)$$



Why? Because:

- ReLU avoids the vanishing gradient problem by keeping gradients high when $x > 0$.
- ReLU activates hidden layers that do not need to produce probabilities better

But these advantages cannot be understood without a scaled up gradient descent and backpropagation, which is how a neural network learns.

NN gradient descent

We begin with a randomly initialized set of weights. Then, we make a pass with some input, and calculate the sum of square differences between the model's prediction, and the true answer.

If we then average all of these "costs" over lots of data, this value is representative of how well or how badly the model is doing.

To find the minima of this cost function $C(w)$, we can perform gradient descent. The general case of a 2-D gradient descent generalizes to n-dimensions, fortunately. We are still computing the negative slope of the point w in the graph $C(w)$ such that we can nudge ourselves to the local minima. Intuitionally,

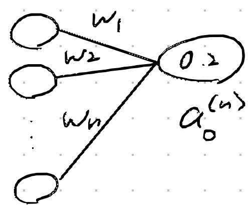
$$\text{if} \quad \bar{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}, \quad \text{then } -\nabla C(\bar{w}) = \begin{bmatrix} 0.31 \\ 0.03 \\ \vdots \\ -0.37 \\ 0.16 \end{bmatrix} \begin{array}{l} w_0 \text{ should increase} \\ w_1 \text{ should increase} \\ \\ w_{n-1} \text{ should decrease} \\ \\ \end{array}$$

and the magnitudes tell us the extent of the decrease/increase.

Backpropagation

NN gradient descent gives us how to calculate the direction towards a minima. But how do we use this to actually change the weights?

Say we have some neuron activation of 0.2 at the output layer, which represents the correct answer.



↳ Each neuron in the previous layer has an activation, which scales the activation of the neuron in the next layer by the weight. In other words, if we want to nudge the weights to change $a_0^{(n)}$, it must be in proportion to the weights.

So, each neuron in a given layer consults all neuron activations in the previous layer for the most bang-for-buck nudges that can be applied to the weights that connect them. Then, once we calculate the "sum nudge" for each neuron in the previous layer, that value then informs nudges that must occur in the previous layer. This process is backpropagation.

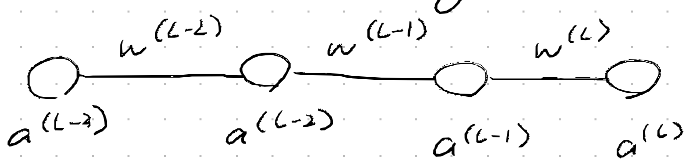
Since computing the gradients for each input would be too intensive, we perform gradient descent in batches.

This ensures the following:

1. Since the batch is more representative of the entire training corpus, it results in less noisy updates.
2. The gradient update is the average of all of the examples in the batch, which allows for parallel computation.

Now, what exactly is the calculus behind backprop?

Suppose a network with 4 layers with one neuron:



And a desired output y

At the last layer, we have $C_0(\ldots) = (a^{(L)} - y)^2$

Let $w^{(L)} a^{(L-1)} + b^{(L)} = z^{(L)}$

then, $a^{(L)} = \sigma(z^{(L)})$

Since we want to calculate changes in cost with respect to weights, we can phrase the problem as such:

$$\frac{\partial C_0 \rightarrow \text{change in cost}}{\partial w^{(L)} \rightarrow \text{tiny nudge}}$$

$$\frac{\partial C_0}{\partial w^{(L)}} = \underbrace{\frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}}_{\text{chain rule}}$$

Given:

$$C_0 = (a^{(L)} - y)^2$$

$$z^{(L)} = w^{(L)} \cdot a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$

$$\frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

$$\frac{\partial a}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

So, $\frac{\partial C_0}{\partial w^{(L)}} = a^{(L-1)} \cdot \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y)$

The full cost derivative is $\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}$

Which is the cost function for $w^{(L)}$ at the last layer.

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix}$$

$$\frac{\partial C}{\partial b^{(L)}} \rightarrow \frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}$$

$$= 1 \cdot \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y)$$

Furthermore,

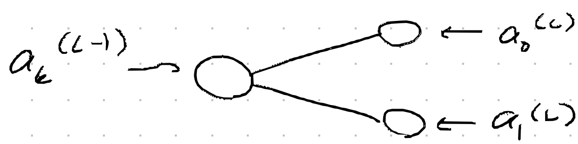
$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}$$

$$= w^{(L)} \cdot \sigma'(z^{(L)})\, 2(a^{(L)} - y)$$

↳ the weights from the last layer affects the weights from the second last layer.

This one neuron layer example generalize to multi-neuron layers, only with additional indices.

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_{L-1}} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j} \cdot \frac{\partial C_0}{\partial a_j^{(L)}}$$

↳ 

OK. How does all this get applied to the transformer?

Large Language Models (and Transformers)

At a high level, we still follow the format of a multilayer perceptron.

input → layers → output.

But now, the input is generally scaled up from vectors to matrices and even n-dim arrays, which we call tensors. Thus the name "tensorflow".

Let's take GPT-3 as an example of this scale.
Its ~175B weights are divided into 27,938 matrices, each of which perform linear transformation with input vectors for various functions:

- Embedding : d_embed · n_vocab
- Key : d_query · d_embed · n_heads · n_layers
- Query : "
- Value : d_value · "
- Output : d_embed · d_value · n_heads · n_layers
- Up-projection : n_neurons · d_embed · n_layers
- Down-projection : d_embed · n_neurons · n_layers
- Unembedding : n_vocab · n_embed.

Embedding (d_embed · d_vocab)

The embedding matrix turns tokens to vectors.

$$W_E = \underbrace{\left[\begin{array}{ccc} v_1^{(1)} & & v_1^{(m)} \\ v_2^{(1)} & & v_2^{(m)} \\ \vdots & \cdots & \vdots \\ v_n^{(1)} & & v_n^{(m)} \end{array}\right]}_{\text{vocabulary}} \quad \text{where } n = 13{,}288$$

embedding

m = number of tokens in dictionary

And similar tokens in the embedding space are closer to one another, which can be calculated by the dot product.

↳ varied according to angle between the two vectors:

- $\theta > 90°$: negative
- $\theta = 90°$: 0
- $\theta < 90°$: positive

One example of populating an embedding matrix is the skip-gram model that is used in Word2Vec.

Input: one-hot encoded vector of the target word
$\in \mathbb{R}^V$

Embedding matrix: Matrix of size $V \times d$, where each row represents the d-dimensional embedding of some target word v.

Output: the embedding vector

Training: given the context of window size k that surrounds the target word in the training data, learn to predict context through backpropagation.

The LLM takes these embeddings as a starting point for each token in the input, and then further encodes the context of the input to each embedding.

At the very end, the last vector in the input will encode the context of the next token. Using an unembedding matrix, we turn that vector back to a probability distribution of the dictionary, so that we can make an informed prediction.

Since the initial probability distribution is not normalized, we run it through a softmax.

"Softmax" derives from the amplifying effects of the exp. on any maxima in the input vector.

$$\underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{\text{Logits}} \rightarrow \underbrace{\begin{bmatrix} \frac{e^{x_1}}{\sum e^{x_n}} \\ \frac{e^{x_2}}{\sum e^{x_n}} \\ \frac{e^{x_3}}{\sum e^{x_n}} \end{bmatrix}}_{\text{Softmax}} \rightarrow \underbrace{\begin{bmatrix} 0.1 \\ 0.2 \\ 0.7 \end{bmatrix}}_{\text{Probabilities}}$$

We throw in some spice to the softmax function to change the softness of the max, so to say.

$$\frac{e^{x_i/t}}{\sum e^{x_n/t}}$$

The greater the temperature, the softer the max.

Attention

As briefly mentioned previously, the word embeddings of each token in the dictionary is constant over all situation. For example, the word "model" has the same context in these two contexts:

"The model walked on stage."

"The model makes next token predictions."

So the goal of the attention blocks in LLMs is to further contextualize vectors given input. More generally, it moves information to and fro across embeddings.

Working example: a fluffy blue creature roamed the verdant forest.
$\overline{E}_4$

Let's think first about the encoding of adjectives to nouns. We call this one "head" of attention.

Imagine the noun "creature" asks the question, "are there any adjectives in front of me?" This is a query, which is represented as a vector. For example, $W_Q \cdot \overline{E}_4 = \overline{Q}_4$.

Imagine again that we can provide answers to such queries. This would take the form $W_K \cdot \overline{E}_4 = \overline{K}_4$.

Both W_Q and W_K are matrices with tunable parameters or weights, and both transform the embedding vector to some smaller space.

$\bar{K}$ and $\bar{Q}$ "match" when they are closely aligned to one another, i.e., a positive dot product.

Dot products of these vectors can then be displayed in a matrix, where each value represents the closeness of their respective key-value pair.

$$
\begin{array}{c|cccc}
\text{Attention Pattern} & \bar{E}_1 \xrightarrow{W_q} \bar{Q}_1 & \bar{E}_2 \xrightarrow{W_q} \bar{Q}_2 & \cdots & \bar{E}_n \xrightarrow{W_q} \bar{Q}_n \\
\hline
\bar{E}_1 \xrightarrow{W_k} \bar{K}_1 & \bar{K}_1 \cdot \bar{Q}_1 & \bar{K}_1 \cdot \bar{Q}_2 & \cdots & \bar{K}_1 \cdot \bar{Q}_n \\
\bar{E}_2 \xrightarrow{W_k} \bar{K}_2 & & & \ddots & \vdots \\
\vdots & & & & \\
\bar{E}_n \xrightarrow{W_k} \bar{K}_n & & & & \bar{K}_n \cdot \bar{Q}_n
\end{array}
$$

When the dot products are great, we say that the respective $\bar{K}$ "attends to" the respective $\bar{Q}$.

Let us then apply a softmax to these values at each column, or query. The resulting normalized values can be considered as weights that show the relevancy of the key and value. Formally put,

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

for numerical stability

we will get back to this later.

One more training detail.

Given some input of length k for training where backprop is performed based on the gradient with respect to the true $k+1^{th}$ true token prediction, we can make training much more efficient by creating subsequences of $k-n$ from $n = 1 \to k-1$.

the → ?

the fluffy → ?

the fluffy blue → ?

⋮

the fluffy blue creature roamed the verdant forest → ?

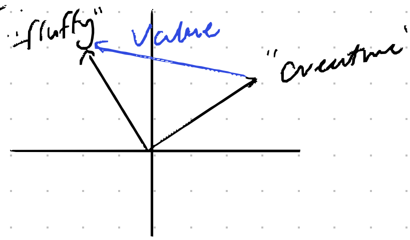
For this additional training step to work, we need the subsequences to be challenging, and not cheat by using the attention from subsequent tokens. Therefore, the area marked with in the attention pattern must be marked as $-\infty$ before softmax such that they become 0. This is called "masking".

Also note that the size of the attention pattern is context_size x context_size, i.e., the permitted number of embeddings for each input.

How do we use attention to actually update the embeddings?

This is where W_V, the value matrix, comes in.

For example, say "creature" attends to "fluffy". By multiplying the W_V to "fluffy", we get a value vector that can be added to "creature".



To perform this computation at scale, we create the value matrix W_V.

	$\bar{E}_1 \downarrow \bar{Q}_1$	$\bar{E}_2 \downarrow \bar{Q}_2$	$\dots$	$\bar{E}_8 \downarrow \bar{Q}_8$
$\bar{E}_1 \to \bar{K}_1 \xrightarrow{W_V} \bar{V}_1$		$A_2 \cdot \bar{V}_1$ +		
$\bar{E}_2 \to \bar{K}_2 \xrightarrow{W_V} \bar{V}_2$		$A_2 \cdot \bar{V}_2$ +		
$\vdots$		$\vdots$ +		
$\bar{E}_n \to \bar{K}_n \xrightarrow{W_V} \bar{V}_n$		$A_n \bar{V}_n$		

$$A_2 \cdot \bar{V}_1 + A_2 \cdot \bar{V}_2 + \dots + A_n \bar{V}_n = \Delta \bar{E}_2$$

where $A_i = \bar{K}_i \cdot \bar{Q}_i$

so $\bar{E}_2 + \Delta\bar{E}_2 = \bar{E}_2'$

This is one "head" of attention. Transformers usually have many more. These ask different questions with different W_Q, different answers with W_K, and different deltas with W_V. GPT-3 uses 96 attention heads, for example.

Lastly, the <u>output matrix</u>. But a small detail before discussion.

The W_V matrix maps one embedding vector to another, meaning that it has dimensions n_embed × n_embed. This is much larger than the matrices W_K and W_Q, which are both size n_embed × n_query_key.

W_k and W_Q maps the embeddings to a much smaller query/key space. In GPT-3, n_embed = 12,288 and n_query_key = 128.

Since we don't want to use a disproportionate amount of compute for calculating value, we perform a low-rank factorization of W_V such that we have $W_{V\downarrow}$ and $W_{V\uparrow}$. $V\downarrow$ represents downprojecting the embedding space to the QK space, and then upprojecting the QK space to the embedding space.

The output matrix is a horizontally stapled-together version of the $W_{V\uparrow}$ matrices, which allows us to only refer to $W_{V\downarrow}$ when we say "value matrix".

We now covered attention. But that is half of the transformer, we should now discuss the multilayer perceptron layers.

Multilayer Perceptron

There has been increasing evidence that the MLP is what stores facts in an LLM.

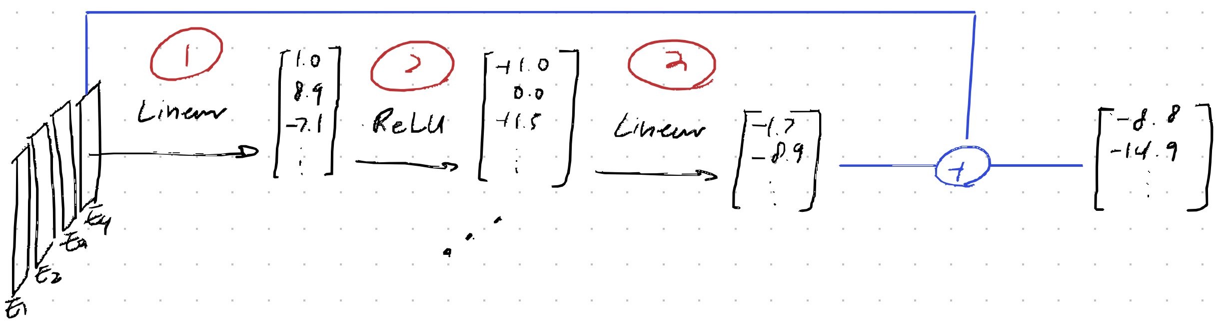
Before we begin, let's introduce a toy example, and some assumptions to go with it.

We want to understand how an LLM might "know" that Michael Jordan plays basketball. Let's assume that a direction in the embedding space of tokens encodes:

1. First name Michael
2. Last name Jordan
3. Basketball.

So, the dot product between some embedding $\overline{E}$ and say, $\overline{\text{Basketball}}$ reveals whether the embedding represents basketball or not. As an example, if $\overline{E} \cdot \overline{\text{Last name Jordan}} = \overline{E} \cdot \overline{\text{First name Michael}}$, we would know that the resulting vector encodes both first name Michael and last name Jordan.

At a high level, an MLP block does the following:



1 This is a matrix multiplication with some matrix. Say, for example, that this matrix looks as such:

$$M = \begin{bmatrix} — \overline{R}_0 — \\ — \overline{R}_1 — \\ \vdots \\ — \overline{R}_n — \end{bmatrix}$$

Suppose further that $\overline{R}_0$ encodes the direction F. N. Michael. Then, the resulting vector will calculate the proximity of the embedding to $\overline{R}_0$.

$$M \cdot \overline{E} = \begin{bmatrix} \overline{R}_0 \cdot \overline{E} \\ \overline{R}_1 \cdot \overline{E} \\ \vdots \end{bmatrix} \quad \begin{cases} \approx 1 & \text{if } \overline{E} \text{ encodes "FN Michael"} \\ \leq 0 & \text{if not.} \end{cases}$$

So each row in M is probing with some "question", and calculates its proximity to $\overline{E}$ [redacted]

There is also the bias vector, which adds or subtracts the final output of 1.

Since M maps E to a higher dim., we will call it $W_\uparrow$.

Let bias be $\overline{B}_\uparrow$.

After the linear transformation, we go through a ReLU, 2

a non-linear function that turns all negative values to 0, and leaves all positive values unchanged.

Why? Because

1. It classifies whether the embedding encodes $\overline{R}_1$ or not as a simple AND gate, on or off.
2. Introducing nonlinearity to the network allows it to encode nonlinear relationships in language.

In detail,

$$\overline{E} \cdot \overline{R}_1 \begin{cases} \approx 1 & \text{if } \overline{E} \text{ encodes } \overline{R}_1 \\ \leq 0 & \text{otherwise} \end{cases} \xrightarrow{\text{ReLU}} = 0 \text{ otherwise.}$$

More simply, think of the final dot product as a neuron. Since it is either activated or deactivated, we don't want negative values.

3 The final linear function downprojects the final dimensions back to that of the embedding space.

$$\underbrace{\begin{bmatrix} — C_1 — \\ \vdots \\ — C_n — \end{bmatrix} \cdot \begin{bmatrix} n_1 \\ \vdots \\ n_m \end{bmatrix}}_{n_1 C_1 + n_2 C_2 + \dots} + \overline{B} = \dots$$

In other words, the first linear detects topics in the embedding and turns those sensitivities to neurons.

Now that we have neurons encoding F.N. Michael and L.N. Jordan, the relevant neurons can either activate or deactivate concepts in $\overline{C_0} - \overline{C_n}$. For example, the vector that encodes $\overline{\text{Basketball}} + \overline{\text{Chicago Bulls}} + \cdots$, which results in the final embedding.

1

In summary, the vectors $\overline{R_0} - \overline{R_{n-1}}$ in W_1 represents directions in the embedding space that will activate the output neurons if they correspond to $\overline{E_i}$. Then, the neurons go through
2
ReLU to add non-linearity and simplicity. Finally, the column vectors $\overline{C_0} - \overline{C_{m-1}}$ represent vectors that
3
will be added to the result, given the active neurons.

Symbolically, $\overline{R_0}$ and $\overline{R_1}$ may detect $\overrightarrow{\text{F.N.M.}}$ and $\overrightarrow{\text{L.N.M.}}$, and $\overline{C_0}$ may be activated by N_0 to encode $\overrightarrow{\text{basketball}}$.