Part I
micrograd

1 Derivatives

Assume all of our functions $f(x)$ take real inputs $x \in \mathbb{R}$. Let's focus on the definition of the derivative at a point.

Definition 1. The derivative function is the function of rise over run, calculated via

$$\frac{rise}{run} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} = \lim_{t \to x} \frac{f(t) - f(x)}{t - x}$$

Intuitively, the derivative is just a means to calculate the slope at a point, which measures how much the function changes based on its input.

Example 1. Let $f(x) = 3x^2 - 4x + 5$. Then $f'(x) = 6x - 4$, and the derivative for this function changes signs depending on x's value,

$$sgn\left(f'\right) = \begin{cases} > 0 & x \in \left(\frac{2}{3}, \infty\right) \\ = 0 & x = \frac{2}{3} \\ < 0 & x \in \left(-\infty, \frac{2}{3}\right) \end{cases}$$

Let $F(x) = x_1 x_2 + x_3$. What happens to $F(x)$ if we increase or decrease the values of each x_i one at a time by some tiny value $h \in \mathbb{R}$? The question we posed is about something called the **partial derivatives**.

Definition 2. The **partial derivative** of a multi-valued function is the derivative with respect to a single variable, treating the other variables as constant scalars.

Example 2. Let $F(x) = x_1 x_2 + x_3$. Observe

$$\frac{\partial F}{\partial x_2} = \lim_{h \to 0} \frac{F(x_1, x_2 + h, x_3) - F(x)}{h} = x_1$$

$$\begin{aligned} \frac{\partial F}{\partial x_3} &= \lim_{h \to 0} \frac{F(x_1, x_2, x_3 + h) - F(x)}{h} \\ &= \lim_{h \to 0} \frac{x_1 x_2 + (x_3 + h) - (x_1 x_2 + x_3)}{h} \\ &= \lim_{h \to 0} \frac{(x_3 + h - x_3)}{h} \\ &= \lim_{h \to 0} \frac{h}{h} \\ &= 1 \end{aligned}$$

The derivative has several useful properties we will use throughout this manuscript.

Proposition 1. *Let f and g be functions defined at x. The derivative is a linear operation, i.e.,*

$$d(f+g)(x) = df(x) + dg(x)$$

Proof. Observe

$$\begin{aligned} d(f+g)(x) &= \lim_{t \to x} \frac{(f+g)(t) - (f+g)(x)}{t - x} \\ &= \lim_{t \to x} \frac{f(t) + g(t) - f(x) - g(x)}{t - x} \\ &= \lim_{t \to x} \frac{f(t) - f(x)}{t - x} + \lim_{t \to x} \frac{g(t) - g(x)}{t - x} \\ &= df(x) + dg(x) \end{aligned}$$

□

Proposition 2. *Let f and g be functions defined at x. The product rule for derivatives is*

$$d(f \cdot g)(x) = df(x) \cdot g(x) + f(x) \cdot dg(x)$$

Proof. Recall the limit splits across multiplication, i.e.,

$$\lim_{x \to a} (f \cdot g)(x) = \lim_{x \to a} f(x) \cdot \lim_{x \to a} g(x)$$

Observe

$$\begin{aligned}
d(f \cdot g)(x) &= \lim_{h \to 0} \frac{(f \cdot g)(x+h) - (f \cdot g)(x)}{h} \\
&= \lim_{h \to 0} \frac{(f \cdot g)(x+h) + 0 - (f \cdot g)(x)}{h} \\
&= \lim_{h \to 0} \frac{(f \cdot g)(x+h) - f(x+h) \cdot g(x) + f(x+h) \cdot g(x) - (f \cdot g)(x)}{h} \\
&= \lim_{h \to 0} \frac{f(x+h)(g(x+h) - g(x)) + (f(x+h) - f(x)) g(x)}{h} \\
&= \lim_{h \to 0} \frac{f(x+h)(g(x+h) - g(x))}{h} + \lim_{h \to 0} \frac{(f(x+h) - f(x)) g(x)}{h} \\
&= \lim_{h \to 0} f(x+h) \cdot \lim_{h \to 0} \frac{(g(x+h) - g(x))}{h} + \lim_{h \to 0} \frac{(f(x+h) - f(x))}{h} \lim_{h \to 0} g(x) \\
&= f(x) \cdot dg(x) + df(x) g(x)
\end{aligned}$$

□

Proposition 3. *Let f and g be functions defined at x. The chain rule for the composition $f \circ g$ states*

$$d(f \circ g)(x) = df(g(x)) \cdot dg(x)$$

Proof. TODO. □

We often need to reproduce the derivative's behavior because won't get the closed form of our functions in practice. Let's explore what happens based on the signs of each value so we can get an intuitive sense of what's going on.

Let's focus on x_1. If $h > 0$, then taking $F(x_1 + h)$ will be less negative regardless of $sgn(x_1)$. Thus, we depend on $sgn(x_2)$ to describe $F's$ output, i.e.,

$$(x_1 + h) x_2 \to \begin{cases} (x_1 + h) x_2 < x_1 x_2 & x_2 < 0 \\ x_1 x_2 < (x_1 + h) x_2 & 0 < x_2 \end{cases}$$

For example, if $x_1 = 2$, $x_2 = -4$, and $h = 0.001$, then

$$\begin{aligned}
(x_1 + h) x_2 &= (2.001)(-4) = -8.004 \\
x_1 x_2 &= 2(-8) = -8
\end{aligned}$$

so we expect the function F to decrease by a small amount, which implies the slope is negative at that point. We can confirm this since

$$\begin{aligned}
\frac{\partial F}{\partial x_1} &= \lim_{h \to 0} \frac{F(x_1 + h, x_2, x_3) - F(x)}{h} \\
&= \lim_{h \to 0} \frac{(x_1 + h) x_2 + x_3 - (x_1 x_2 + x_3)}{h} \\
&= \lim_{h \to 0} \frac{(x_1 + h - x_1) x_2}{h} \\
&= \lim_{h \to 0} \frac{h x_2}{h} \\
&= x_2,
\end{aligned}$$

which means the partial derivative for x_1 depends solely on x_2. One can confirm $\frac{\partial F}{\partial x_2} = x_1$ as well.

What about x_3? Increasing x_3 by a small amount $h > 0$ will only result in a rise via the same increase, i.e.,

$$F(x_1, x_2, x_3 + h) - F(x_1, x_2, x_3) = x_3 + h - x_3 = h$$

which implies the slope is one at that point. Again, we can confirm this analytically since

$$\begin{aligned}
\frac{\partial F}{\partial x_3} &= \lim_{h \to 0} \frac{F(x_1, x_2, x_3 + h) - F(x)}{h} \\
&= \lim_{h \to 0} \frac{x_1 x_2 + (x_3 + h) - (x_1 x_2 + x_3)}{h} \\
&= \lim_{h \to 0} \frac{(x_3 + h - x_3)}{h} \\
&= \lim_{h \to 0} \frac{h}{h} \\
&= 1
\end{aligned}$$

We will soon see that neural networks are just functions composed of tens of thousands of terms, and most algorithms involving neural networks require calculating **partial derivatives**2. However, no one who works with neural networks actually writes out the derivative expression for something so massive. This means we need to numerically approximate them. Computer memory systems can only store so many digits accurately, so we have to keep these limitations in mind. The neural network functions are also too large to feasibly compute the derivative analytically, so we need some data structure that can store these massive expressions and compute on them. These computation and optimization issues plagued mathematicians for decades until recently with the improvement of modern processors. Now, we can comfortably create and train "small" neural nets on computers from the early 2010s.

2 Forward and Backward Propagation

Imagine you have a equation such as

$$L = w_6 (w_3 + w_1 w_2)$$

for $w_i \in \mathbb{R}$. This L is just a composition of many operations, i.e., let f_+ be the addition function and $f_\cdot$ be the multiplication function, then

$$L = f_\cdot (w_6, f_+ (w_3, f_+ (w_1, w_2)))$$

We can further break this down into a graph structure, in which the w_i and their corresponding operations are the vertices, and the edges are directed arrows that identify the operation's inputs. With this graph structure, we can visual L as a forward pass.

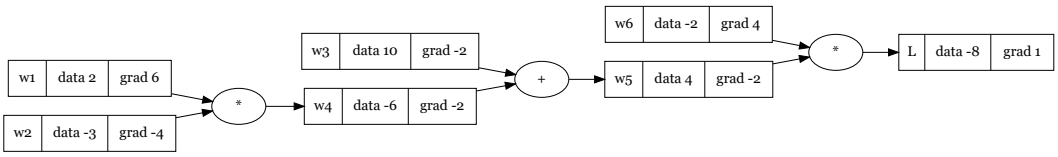


Figure 1: Operational graph structure of L representing a forward pass.

What if we started from L and then traversed backwards to its inputs? One way to do this is via a traversal method called **back-propagation**, in which we calculate the **gradient** at every intermediate value, which is the derivative of w_i with respect to L.

Definition 3. The **gradient** is the partial derivative of L with respect to w_i, denoted by $\frac{\partial L}{\partial w_i}$.

For neural networks, this function L contextually represents a **loss function**, and its inputs w_i are called **weights**. Our goal is to understand how these weights impact the loss function by calculating the gradient at each point.

How does w_3 influence L? Does it have a positive, neural, or negative impact? The gradient can help us answer this question by observing its sign. If $h > 0$ is a very small number, we can calculate the gradient at $w_3$using the definition of the derivative and treating all $w_i \neq w_3$ as constants, i.e.,

$$\begin{aligned}
\frac{\partial L}{\partial w_3} &= \lim_{h \to 0} \frac{f.\left(w_6, f_+\left(\left(w_3 + h\right), f_+\left(w_1, w_2\right)\right)\right) - f.\left(w_6, f_+\left(w_3, f_+\left(w_1, w_2\right)\right)\right)}{h} \\
&= \lim_{h \to 0} \frac{w_6\left(\left(w_3 + h\right) + \left(w_1 + w_2\right)\right) - w_6\left(w_3 + \left(w_1 + w_2\right)\right)}{h} \\
&= \lim_{h \to 0} \frac{w_6\left(w_3 + h\right) + w_6\left(w_1 + w_2\right) - w_6 w_3 - w_6\left(w_1 + w_2\right)}{h} \\
&= \lim_{h \to 0} \frac{w_6\left(w_3 + h\right) - w_6 w_3}{h} \\
&= \lim_{h \to 0} \frac{w_6\left(w_3 + h - w_3\right)}{h} \\
&= w_6
\end{aligned}$$

What about the gradient at an implicit variable such as $w_4 = f_+\left(w_1, w_2\right)$? Then we have to use substitution, i.e.,

$$\frac{\partial L}{\partial w_4} = \frac{\partial L}{\partial f_+\left(w_1, w_2\right)} = \frac{\partial L}{\partial\left(w_1 + w_2\right)}$$

since the derivative is linear by 1. Observe this final output is the same as taking the derivative of L with respect to $w_1 + w_2$. Now, let's calculate it analytically,

$$\begin{aligned}
\frac{\partial L}{\partial w_4} &= \lim_{h \to 0} \frac{f.\left(w_6, f_+\left(w_3, \left(w_4 + h\right)\right)\right) - f.\left(w_6, f_+\left(w_3, w_4\right)\right)}{h} \\
&= \lim_{h \to 0} \frac{f.\left(w_6, f_+\left(w_3, f_+\left(w_1, w_2\right) + h\right)\right) - f.\left(w_6, f_+\left(w_3, f_+\left(w_1, w_2\right)\right)\right)}{h} \\
&= \lim_{h \to 0} \frac{w_6\left(w_3 + w_1 + w_2 + h\right) - w_6\left(w_3 + w_1 + w_2\right)}{h} \\
&= \lim_{h \to 0} \frac{w_6\left(w_3 + w_1 + w_2\right) + w_6 h - w_6\left(w_3 + w_1 + w_2\right)}{h} \\
&= \lim_{h \to 0} \frac{w_6 h}{h} \\
&= w_6
\end{aligned}$$

2.1 Chain Rule

We can also rewrite these expressions using the chain rule since L is just a composition of functions. This will also save us a lot of computational time. Let's try this again from our first example. Referencing 1, we can write the derivative using the chain rule as

$$\begin{aligned}
\frac{\partial L}{\partial w_3} &= \frac{\partial L}{\partial w_5} \cdot \frac{\partial w_5}{\partial w_3} \\
&= \frac{\partial\left(w_5 w_6\right)}{\partial w_5} \cdot \frac{\partial\left(w_3 + w_4\right)}{\partial w_3} \\
&= w_6 \cdot 1 \\
&= w_6
\end{aligned}$$

This highlights something important, which is that addition operations have no bearing on L's outcome since their derivative is always going to be one.

Lemma 1. *The partial derivative of an addition operation with respect to any of its child vertices is always one.*

Proof. Let $\left(\{w_i\}_{i\in V}, +\right)$ be the child edges of an addition operation vertex. WLOG, choose w_j for some vertex in the set of child vertices. Observe since w_i are scalars and differentiation is linear,

$$\frac{\partial}{\partial w_j}\sum_{i\in V} w_i = \sum_{i\in V}\frac{\partial w_i}{\partial w_j} = \frac{\partial w_j}{\partial w_j} + \sum_{i\in V; i\neq j}\frac{\partial w_i}{\partial w_j} = 1$$

□

Lemma 2. *The partial derivative of a multiplication operation with respect to any of its child vertices is the product of the other factors excluding the child vertex.*

Proof. Let $\left(\{w_i\}_{i\in V}, \cdot\right)$ be the child edges of a multiplication operation vertex. WLOG, choose w_j for some vertex in the set of child vertices.

Since w_i are scalars, we can use the product rule from 2 by fixing w_j and treating it as $f(x)$, and treat the product of the others as $g(x)$. Observe,

$$\begin{aligned}
\frac{\partial}{\partial w_j}\left(\prod_{i\in V} w_i\right) &= \frac{\partial}{\partial w_j}\left(w_j \prod_{i\in V; i\neq j} w_i\right) \\
&= \frac{\partial w_j}{\partial w_j}\prod_{i\in V; i\neq j} w_i + w_j \sum_{k\in V; k\notin\{i,j\}}\left(\frac{\partial w_k}{\partial w_j}\prod_{i\in V; i\notin\{j,k\}} w_i\right) \\
&= 1\cdot\prod_{i\in V; i\neq j} w_i + w_j \sum_{k\in V; k\notin\{i,j\}}\left(0\cdot\prod_{i\in V; i\notin\{j,k\}} w_i\right) \\
&= \prod_{i\in V; i\neq j} w_i
\end{aligned}$$

□

Proposition 4. *Let L be the outcome vertex of a forward propagation consisted entirely of $(+)$ and $(\cdot)$ operations. Let (V, E) be the induced connected graph of L with no cycles. If w_i is n operations from L, then for k $(\cdot)$-based operations and $n-k$ $(+)$-based operations,*

$$\frac{\partial L}{\partial w_i} = (1)^{n-k}\prod_{m\in\mathcal{M}(w_i)}\prod_{j\in\mathcal{O}(m)-\{w_i\}} w_j$$

where $\mathcal{M}$ is the set of all multiplication operations involving w_i, and $\mathcal{O}$is the set of all operands for a given multiplication operation.

Proof. **TODO:** Split the vertex set into vertices that are children of a $(+)$ and $(\cdot)$ operation. Then use the previous proposition to get the 1s. Finally, the pigeonhole principle should apply to get the non-1s. □

The point of this is to highlight that in a graph context, the addition operations contribute nothing to the gradients, and we only need to just collect all of the non-w_i leaf factors. These propositions are essentially saying we can calculate the gradients at each vertex via back-propagation efficiently by ignoring the $(+)$ operations and taking the product of the derivative of vertices that are children of $(\cdot)$ operations.

Example 3. Let $L = w_6(w_3 + w_1w_2)$ to calculate $\frac{\partial L}{\partial w_1}$.

We need to construct the operation and operand sets based on the graph, i.e.,

$$\mathcal{M}(w_1) = \{w_1w_2, w_6(w_3 + w_1w_2)\}$$

Notice $|\mathcal{M}(w_1)| = 2$. For each multiplication operation in $\mathcal{M}$, then

$$\begin{aligned}
\mathcal{O}(w_1w_2) &= \{w_1, w_2\} \\
\mathcal{O}(w_6(w_3 + w_1w_2)) &= \{w_6, (w_3 + w_1w_2)\}
\end{aligned}$$

Then subtracting any operation involving w_1 reduces each to

$$\mathcal{O}(w_1 w_2) - \{w_1\} = \{w_2\}$$
$$\mathcal{O}(w_6(w_3 + w_1 w_2)) - \{w_1\} = \{w_6\}$$

Finally, multiply all the remaining operands yields the final result.

$$\frac{\partial L}{\partial w_1} = w_2 w_6$$

These gradients are useful because they inform us how the individual vertices affect the final output. For example, if one wants to increase the output, then one needs to adjust the inputs in the direction of the gradient, and vice versa. For neural networks, we are most interested in the derivative of the output with respect to the initial weights since they are going to be changing as a part of the optimization process. We will eventually introduce the concept of a loss function that calculates the accuracy of our output, and we will be back-propagate with respect to that accuracy with the intention of increasing it.

3 Activation functions

Activation functions are non-linear functions with a very small range (distance-wise). The idea is to squash outputs to desired ranges, e.g., classifiers prefer a normalized probability range of $[0, 1]$. It's also necessary to have non-linear functions that allow our model's function outputs to take on different shapes. This is what is known as "learning."

One common activation function is the hyperbolic tangent function because it calculates a unit length, i.e., $\tanh(x) \in (-1, 1)$.

Part II
makemore via ngrams

This covers the character level ngram model that predicts the next character in a sequence given a fixed character sequence of size $n - 1$.

Definition 4. An **ngram** is a sequence of n adjacent symbols in particular order.

4 Dataset

Collected data is a sample output of some unknown probability distribution, and machine learning's goal is to recreate that distribution function using that data. The more data we have, the easier it is to recreate that model. That's why very complicated problems modeled with neural networks require millions of data pieces.

Our dataset is collection of names such that each name is a data piece. A data piece in this context is an existing sequence of characters from a given alphabet, but importantly, this sequence of characters is intentional and not completely random since this data comes from an unknown distribution. A name like "Isabella" represents multiple character sequence examples, namely,

- "i" is very likely to come first in a **name**
- "s" is very likely to come after "i"
- "a" is very likely to come after "is"
- ...
- "l" is very likely to come after "isabel"

- "a" is very likely to come after "isabell"
- "isabella" is very likely to be the whole word.

This is why representation matters, because your model depends on the data properly representing the distribution since it is impossible in most cases to collect all the data. This is why the phrase "garbage in, garbage out" matters a lot in the machine learning world, because a random character sequence that is not a name like "aywjtpe" could taint the model's learning.

5 Bigram Language Models

Given a one token sequence, what token is likely to follow? We can try to answer this question empirically via counting at the character level, where names are consisted entirely of characters or letters in our alphabet.

5.1 Graph representation

We'll construct a directed graph using characters as the vertices and draw an edge (v_1, v_2) if $v_1 v_2$ appears as a bigram. We will also prepend each word in the dataset with a start and end token to denote if/when a character starts the word or ends the word. Then the adjacency matrix A will be the $|V| \times |V|$ matrix in which $A[i,j]$ means (v_i, v_j) forms an edge. Observe there can be multiple edges due to repeated sequences, so E is a multi-set. Define

$$A[i,j] = \begin{cases} \sum_{(v_i,v_j)\in E} I_e\left((v_i,v_j)\right) & (v_i,v_j) \in E \\ 0 & (v_i,v_j) \neq E \end{cases}$$

where e is the support of E and I is the indicator function on e. So $A[i,j]$ counts the number of times (v_i, v_j) appears in the character.

Example 4. Given the name "emma", we'll construct "<S>emma<E>". The vertex set will be 0

$$V = \{< S >, e, m, a, < E >\}$$

The edge (multi)set of bigrams will be

$$E = \{(< S >, e), (e, m), (m, m), (m, a), (a, < E >)\}$$

with

$$A = \begin{array}{c|ccccc} & m & e & a & < S > & < E > \\ \hline m & 1 & 0 & 1 & 0 & 0 \\ e & 1 & 0 & 0 & 0 & 0 \\ a & 0 & 0 & 0 & 0 & 1 \\ < S > & 0 & 1 & 0 & 0 & 0 \\ < E > & 0 & 0 & 0 & 0 & 0 \end{array}$$

and graph shown in 2. Notice we can quickly identify if there are any self-loops in the graph if there are non-zero entries along the diagonal. Also, observe the end token <E> is never going to be the start of a character bigram, so $A[< E >, :] = 0$ for all other characters. Likewise, the start token <S> will never be end of a character bigram since all names start with <S> by construction, which means $A[:, < S >] = 0$ for all other characters. Finally, for the bigram construction, the column sum is the same as the row sum.

TODO: Prove the previous stated facts

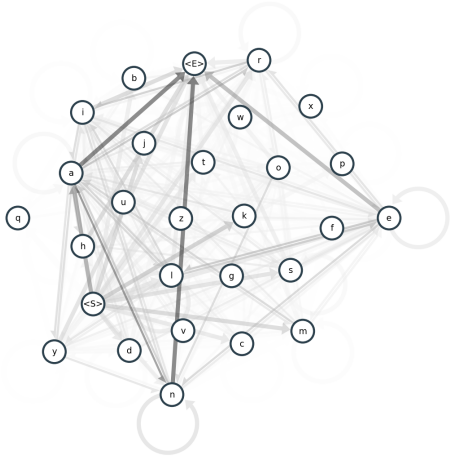


Figure 3: Directed graph of bigram sequences for entire dataset. The arrow density indices the frequency of character sequences (v_1, v_2), which implies it is darker for more frequent and lighter for less frequent. From this, we can deduce "n" and "a" are most often followed by a "<E>", indicating they are the last character in most data pieces in our given dataset.

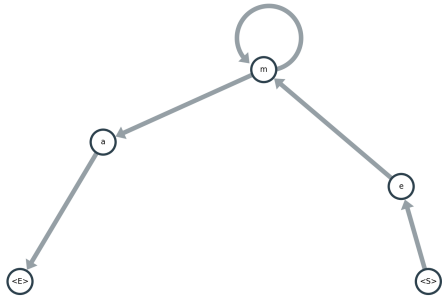


Figure 2: Directed graph of bigram sequences for "emma"

The character level bigram graph for the entire dataset of ~32K names (at the time of this writing) is found at 3.

In Natural Language Processing (NLP) literature, we typically denote these special characters with brackets, but that is cumbersome and wastes computational space. This means we can actually use a single token as as the terminal token for both endpoints.

5.2 Formalizing Bigram Probabilities

Let's restate our assumptions so far. We assume all entries in our data set are composed of a sequence of tokens. The **vocabulary**, V, is a collection of symbols or tokens we can use to construct sequences. We need one vocabulary to represent vocal sounds and to represent words.

Example 5. The English alphabet can be represented with 26 letters and stop token(s), i.e.,

$$V = \{a, b, c, ..., x, y, z\} \cup T$$

in which T is one (or two) special terminal token(s).

We also assume every sequence has a stop that can only appear at the end. We also assume we can see any token multiple times in succession, and each token is conditioned on only the previous token.

Let X denote our assumptions and v_i be the statement token $v \in V$ appears in the ith position. Then an entire sequence is a conjugate statement of v_i's, namely

$$P\left(\prod_{i=1}^{n} v_i | X\right) = \prod_{i=1}^{n} P\left(v_{i+1} | v_i X\right)$$

since each bigram is pair-wise independent, i.e.,

$$P\left(v_{i+1} | v_i \prod_{j=1}^{i-1} v_j X\right) = P\left(v_{i+1} | v_i X\right)$$

which means we only look to a single previous token to predict the next. Now, this only defines a probability distribution for a sequence of n tokens. Our goal is to construct a probability distribution that generates sequences regardless of length and allows us to compare the probabilities of different sequences against each other. Recall our assumption every sequence has to stop, which means $P\left(T | v_i X\right)$ must be very high for some tokens, otherwise the sequence could possibly be infinite depending on the data. For example, think of a dataset that only has names with repeated tokens. How would you know when to stop?

5.3 Calculating Bigram Probabilities via Maximal Likelihood Estimation

Assume we have a sequence of length $n \in \mathbb{N}$, then we can induce a graph $G\left(V, E\right)$ with vertex set V and edge set E. We can write the probability of getting the sequence using bigrams,

$$\begin{aligned} P\left(\prod_{i=1}^{n} v_i | X\right) &= \prod_{i=1}^{n} P\left(v_{i+1} | v_i X\right) \\ &= \prod_{i=1}^{n} P\left(v_i | X\right)^{I(v_i)} \prod_{i=1}^{n} \prod_{j=1}^{n} P\left(v_j | v_i X\right)^{C(v_j, v_i)} \end{aligned}$$

in which $C\left(v_i, v_j\right)$ is the count of how often sequence $v_i v_j$ appears in the entire sequence, and

$$I\left(v\right) = \begin{cases} 1 & (< S >, v) \in E \\ 0 & (< S >, v) \notin E \end{cases}$$

i.e., we can split up the sequence into pair-wise sequences that can appear. Notice this has more terms than we actually have in the original sequence because we are accounting for pair-wise sequences that do not appear since $C\left(v_j, v_i\right) = 0$ implies $(v_i, v_j) \notin E$ for some i and j.

Next, we will take the logarithm on both sides since the log of the product is the sum of the logs. Observe,

$$\begin{aligned} \log\left(P\left(\prod_{i=1}^{n} v_i | X\right)\right) &= \log\left(\prod_{i=1}^{n} P\left(v_i | X\right)^{I(v_i)} \prod_{i=1}^{n} \prod_{j=1}^{n} P\left(v_j | v_i X\right)^{C(v_j, v_i)}\right) \\ &= \sum_{i=1}^{n} \log\left(P\left(v_i | X\right)^{I(v_i)}\right) + \sum_{i=1}^{n} \sum_{j=1}^{n} \log\left(P\left(v_j | v_i X\right)^{C(v_j, v_i)}\right) \\ &= \sum_{i=1}^{n} I\left(v_i\right) \log\left(P\left(v_i | X\right)\right) + \sum_{i=1}^{n} \sum_{j=1}^{n} C\left(v_j, v_i\right) \log\left(P\left(v_j | v_i X\right)\right) \end{aligned}$$

We have the likelihood that this sequence occurs, so now we would like to know when this is likelihood is maximal, i.e., we want to maximize

$$\log\left(P\left(\prod_{i=1}^{n} v_i | X\right)\right) = \sum_{i=1}^{n} I\left(v_i\right) \log\left(P\left(v_i | X\right)\right) + \sum_{i=1}^{n} \sum_{j=1}^{n} C\left(v_j, v_i\right) \log\left(P\left(v_j | v_i X\right)\right)$$

subject to

$$\sum_{j=1}^{n} P\left(v_j|v_iX\right) = 1, \forall i$$

Proposition 5. *Assume we have a sequence of length $n \in \mathbb{N}$ with given assumptions X. The maximum of the bigram language model*

$$\log\left(P\left(\prod_{i=1}^{n} v_i|X\right)\right) = \sum_{i=1}^{n} I\left(v_i\right)\log\left(P\left(v_i|X\right)\right) + \sum_{i=1}^{n}\sum_{j=1}^{n} C\left(v_j, v_i\right)\log\left(P\left(v_j|v_iX\right)\right)$$

subject to

$$\sum_{j=1}^{n} P\left(v_j|v_iX\right) = 1, \forall i$$

occurs when

$$P\left(v_j|v_iX\right) = \frac{C\left(v_j, v_i\right)}{\sum_{j=1}^{n} C\left(v_j, v_i\right)}$$

Proof. Define

$$f_i \coloneqq \sum_{j=1}^{n} P\left(v_j|v_iX\right) - 1$$

then $f_i = 0$ and we can use the Lagrange multiplier

$$\mathcal{L} = \sum_{i=1}^{n} I\left(v_i\right)\log\left(P\left(v_i|X\right)\right) + \sum_{i=1}^{n}\sum_{j=1}^{n} C\left(v_j, v_i\right)\log\left(P\left(v_j|v_iX\right)\right) + \lambda \cdot f_i$$

and find the critical numbers for each $\lambda_i \in \lambda$ by taking the partial derivatives w.r.t $P\left(v_j|v_iX\right)$ for a fixed i, j. Observe

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial P\left(v_j|v_iX\right)} &= \frac{\partial}{\partial P\left(v_j|v_iX\right)}\left(\sum_{i=1}^{n} I\left(v_i\right)\log\left(P\left(v_i|X\right)\right) + \sum_{i=1}^{n}\sum_{j=1}^{n} C\left(v_j, v_i\right)\log\left(P\left(v_j|v_iX\right)\right) + \lambda \cdot f_i\right) \\
&= 0 + \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{C\left(v_j, v_i\right)}{P\left(v_j|v_iX\right)} + \lambda \cdot \frac{\partial f_i}{\partial P\left(v_j|v_iX\right)} \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{C\left(v_j, v_i\right)}{P\left(v_j|v_iX\right)} + \lambda \cdot \frac{\partial\left(\sum_{j=1}^{n} P\left(v_j|v_iX\right) - 1\right)}{\partial P\left(v_j|v_iX\right)} \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{C\left(v_j, v_i\right)}{P\left(v_j|v_iX\right)} + \lambda \cdot \left(\sum_{j=1}^{n} 1\right) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{C\left(v_j, v_i\right)}{P\left(v_j|v_iX\right)} + \sum_{i=1}^{n}\left(\lambda_i \sum_{j=1}^{n} 1\right)
\end{aligned}$$

since $C\left(v_j, v_i\right)$ and λ_i are scalar constants. Most of these summation terms go away since $P\left(v_j|v_iX\right)$ is fixed for some i, j, so we have

$$0 = \frac{\partial \mathcal{L}}{\partial P\left(v_j|v_iX\right)} = \frac{C\left(v_j, v_i\right)}{P\left(v_j|v_iX\right)} + \lambda_i \implies P\left(v_j|v_iX\right) = -\frac{C\left(v_j, v_i\right)}{\lambda_i}$$

Using our constraint, observe that for every i,

$$1 = \sum_{j=1}^{n} P\left(v_j|v_iX\right) = -\sum_{j=1}^{n} \frac{C\left(v_j, v_i\right)}{\lambda_i} \implies -\lambda_i = \sum_{j=1}^{n} C\left(v_j, v_i\right)$$

since λ_i is independent of j. Finally, substitution yields

$$P\left(v_j|v_iX\right) = -\frac{C\left(v_j, v_i\right)}{\lambda_i} = \frac{C\left(v_j, v_i\right)}{\sum_{j=1}^{n} C\left(v_j, v_i\right)}$$

□

We can calculate the probability distribution for each token by normalizing the rows across our adjacency matrix with respect to the raw sum. In other words, for some fixed token v_i, we can calculate the probability v_j follows v_i for every $v_j \in V$,

$$P\left(v_j|v_i\right) = \frac{C\left(v_i, v_j\right)}{\sum_{v_j \in V} C\left(v_i, v_j\right)} = \frac{A\left[i, j\right]}{\sum_j A\left[i, j\right]}$$

Let P denote the matrix constructed from A by performing these normalized row-wise operations, i.e.,

$$P\left[i, j\right] = \frac{A\left[i, j\right]}{\sum_j A\left[i, j\right]}$$

In torch, we calculate these by first taking the row sums of A to get a column vector of size $n \times 1$, then taking the broadcasted division operation to effectively calculate

$$P = \frac{A}{\sum_j A}$$

The elements of this probability matrix can be thought of as parameters for the model because they both give and summarize the statistics of our bigrams.

5.4 Training the model

We effectively trained the model by using our data to construct these parameter probabilities, and we can store this information in a new matrix that is constructed from the normalized rows of our adjacency matrix. This is effectively a closed form solution. How do we know our model has learned anything? Assuming each $v \in V$ is uniformly likely, the principle of indifference implies

$$P\left(v|X\right) = \frac{1}{|V|}$$

If for any $v_j \in V$ satisfies

$$\frac{1}{|V|} < P\left(v_j|v_i\right),$$

then we know the model "learned" something because it assigned a higher probability than just sampling uniformly.

Remark 1. NOTE: The author states verbatim, "If you have a very good model, you'd expect that these [bigram] probabilities should be near one, because that means that your model is correctly predicting what's going to come next." I'm not sure why that's the case, because it's not clear **what specifically** needs to be near one.

5.5 Sampling from the bigram model

The algorithm to sample from a bigram model is as follows

1. Initialize storage with the token and set current token to the start token
2. Sample from $M \sim MultiNorm\left(p = P\left[\text{current token}, :\right], n = 1\right)$ to get the next current token
3. Update the storage with the current token
4. If current token is the start token, break

6 Negative Log-Likelihood Loss

How do we evaluate our model? Recall loss is a single, numerical, non-negative value that tells us how good our model does. Ideally, we want our loss to be low in the absolute value sense. A common loss function for models that use the maximum likelihood estimation is the negative log-likelihood.

Let N be the number of data pieces in our dataset D, E_k be the edges of tokenized document k, and $P[i,j]$ be the probability from $v_i \to v_j$ with our assumptions stored in our computed matrix P from adjacency matrix A, then

$$\mathit{Likelihood}(D) = \prod_{(v_i,v_j)\in\{E_k\}_{k=i}^{N}} P[i,j]$$

Recall products of many numbers that have a wide range of magnitudes can be inaccurate because the result can accumulate floating point arithmetic errors in a phenomena described as "computationally unstable." Our probabilities can vary from very small to almost one, so to mitigate this, we take the logarithm of the likelihood to convert the product into a sum of logarithms, i.e.,

$$\log(\mathit{Likelihood}(D)) = \sum_{(v_i,v_j)\in\{E_k\}_{k=i}^{N}} \log(P[i,j])$$

In practice, we calculate this by iterating over all documents, calculating each bigram and its respective log-probability, then accumulating the sum.

Recall logs are monotonic and continuous on $\mathbb{R}$, which means they preserve the sign of their inputs. Also, recall $\log((0,1]) = (-\infty, 0]$, so $\log(P[i,j]) \in (-\infty, 0]$. Loss is a non-negative value we want to minimize, so we take the negative log-likelihood to ensure our resultant value is in $[0,\infty)$, i.e.,

$$NLL = -\log(\mathit{Likelihood}(D)) = -\sum_{(v_i,v_j)\in\{E_k\}_{k=i}^{N}} \log(P[i,j])$$

6.1 Preventing infinite loss via smoothing

How do we prevent $P[i,j] = 0$ and thus prevent our loss from taking infinite values? Smoothing is a process that makes some global change to the inputs in order to ensure computationally stable results. In our case, we have to take a look at our data to account for and prevent situations that can cause $P[i,j] = 0$. For example, what if we wanted to calculate the NLL for a data piece that contains a bigram we do not have in our dataset, such as "Andrej" or even "Andrejq"? Recall $A[i,j] = 0$ if $(v_i, v_j) \notin E$. Our current model has

$$P[v_j, v_q] = \frac{A[v_j, v_q]}{\sum A[v_j, :]} = \frac{0}{\sum A[v_j, :]} = 0$$

which means $\log(P[v_j, v_q]) \to \infty$ and our loss will be infinite. However, if we offset all values of A by $A + s$ for some $s \in \mathbb{N}$, then $0 < A[i,j]$ and $0 < P[i,j]$ for all i, j. This s is called a **smoothing constant**. Smoothing will unfortunately slightly alter the resultant probabilities by making them more uniform, but it will help the model account for unknown values when we sample.

Why does the smoothing increase the uniformity of the probabilities? Observe

$$P[v_j, v_q] = \lim_{s\to\infty} \frac{A[v_j, v_q] + s}{\sum_j (A[v_j, :] + s)} = \lim_{s\to\infty} \frac{\frac{A[v_j,v_q]+s}{s}}{\frac{\sum_j A[v_j,:]+sj}{s}} = \lim_{s\to\infty} \frac{\frac{A[v_j,v_q]}{s}+1}{\frac{\sum_j A[v_j,:]}{s}+j} = \frac{1}{j}$$

which means v_q satisfies the principle of indifference, implying every token has an equal chance of being chosen.

7 High Dimensionality Problem

If we wanted to model trigrams, quadgrams, or even ngrams, then we will quickly discover constructing and storing the matrix P will be very costly since P will increase to an$|V|^n$ square matrix. We need to use a

different approach. Given an input dataset and a labeled dataset that provides that correct outcome for a model to predict, our goal is to construct a model that can produce the same loss.

Part III
makemore via neural network

A neural network model is a connected graph s.t. each neuron (or vertex) performs linear transformations via randomized weight vectors and matrices, i.e., $W_i x + b_i$. For classification problems, these transformations are passed to an activation function that calculates a probability distribution for the next output, which is compared against the true output to calculate the loss.

We will be using the same character level dataset D constructed from the same vocabulary

$$V = \{a, b, \ldots, y, z\} \cup \{< S >\}$$

which is all of the letters in the standard English alphabet plus one start/stop token. This is only for consistency, as the actual tokens do not matter, e.g., we could have chosen to use words in a sentence as tokens.

8 One-Hot Encoding as a lookup

Recall a **vocabulary** is a countable collection of unique symbols called tokens we use to build sequences. We need one vocabulary to represent words in the form of an alphabet. We can setup a one-to-one correspondence with the naturals and assign each token in a vocabulary its own number called an index. For example, if a vocabulary has only three tokens $V = \{a, b, c\}$, we can take $f(V) = \{0, 1, 2\}$. Then the sequence such as $v = [b, c, a]$ could be mapped to sequence of indices

$$f(v) = [1, 2, 0].$$

Definition 5. An index set starting at 1 is called 1-indexed, whereas an index starting at 0 is called 0-indexed.

One-Hot Encoding represents each token by the standard basis on the Euclidean plan $\mathbb{R}^n$, where n is the vocabulary size, $|V|$ and each token is indexed so that e_i assigns each token its own basis vector. Each row of this encoding has only one non-zero element since it is made up entirely of basis row vectors $\left[\{e_i\}_{i\in\{1,\ldots,n\}}\right]$, which implies the matrix multiplication acts as a lookup table, e.g., given a matrix $W \in \mathbb{R}^{r\times m}$ and the kth row of an encoding X,

$$X[k,:] \cdot W = e_i \cdot W = W[i,:]$$

produces the ith row of W. This trick of using a basis vector to pull out a particular row of a matrix is at the core of how our neural network will work.

Example 6. Let $V = \{a, b, c\}$ be our toy set, then

$$X = OHE \circ f(v) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \in \mathbb{R}^3$$

Given $W \in M_{10\times 27}(\mathcal{N}(0,1))$, then observe $X \cdot W$ pulls out rows (0-indexed) 1,2, and 0 from matrix W in that order.

$$X \cdot W = \begin{bmatrix} e_1 \\ e_2 \\ e_0 \end{bmatrix} \cdot W = \begin{bmatrix} W[1,:] \\ W[2,:] \\ W[0,:] \end{bmatrix} = W[[1,2,0],:] \in M_{3\times 27}(\mathcal{N}(0,1))$$

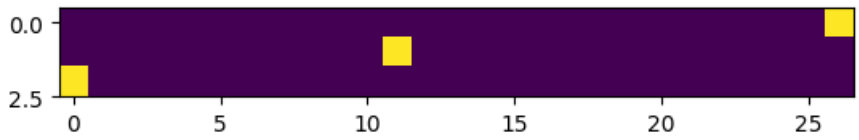


Figure 4: A $(n-1, |V|)$ row of a one-hot encoding from an encoded 4-gram index matrix. Observe each row in this matrix only has one non-zero entry.

For an ngram model, we have a matrix

$$W \in M_{((n-1),|V|,|V|)}\left(\mathcal{N}\left(0,1\right)\right)$$

in which each dimensional slice represents the probability distribution for the next token in the sequence. We also have an index matrix $x \in \mathbb{N}_{(m,n-1)}$ s.t. m is the total number of $n$grams constructed from our dataset. The one-hot encoding matrix $X \in \mathbb{N}_{(m,n-1,|V|)}$ contains m rows of $(n-1, |V|)$ matrices. See 4 for an example.

The resultant matrix $X \cdot W$ can be calculated using **Einstein Summation Notation**, which allows us to quickly reduce the operations of higher dimensional matrix multiplications using only the indices. Observe

$$X \cdot W \in \mathbb{N}_{(m,n-1,|V|)} \cdot M_{((n-1),|V|,|V|)} = X^{m}_{(n-1,|V|)} W^{(n-1,|V|)}_{|V|} \in M_{(m,|V|)}$$

so we are essentially treating the $(n-1, |V|)$ rows and columns of X and W as we normally would when multiplying two dimensional matrices.

8.1 Optimizing the lookup

The basis vectors are cheap to generate, but they do come at a cost of storage of all of those zeros. Let's see how we can optimize this.

TODO: See code and then flesh it out

9 Softmax Activation

Let n be the total number of bigrams generated from our dataset, D. Let xs be the $1 \times n$ index vector of the first character in each bigram, y be the $1 \times n$ index vector of the second character in each bigram. Let X be xs's one-hot encoded matrix representation of size $n \times |V|$. Let W be a matrix of size $|V| \times |V|$ with entries initially sampled from a standard normal distribution $\mathcal{N}\left(0,1\right)$. Our goal is to come up with a function such that $X \cdot W$ produces a matrix P that allows us that predict the next output y, i.e.,

$$X \cdot W \rightarrow ??? \rightarrow P$$

Each element $W\left[i,j\right] \in \mathcal{N}\left(0,1\right)$ can take on small values of different signs such as -0.003 or 3.4. So we can think of each row of

$$X \cdot W = \left[\{e_i\}\right] = W\left[\{i\}, :\right]$$

as the matrix of log-counts for each token token $v_j \in V$ that follows each given v_i. These are called **logits** in the literature. Next, to get the actual counts, we take the inverse operation of the logarithm on the operation to get the frequency matrix, i.e.,

$$A\left[\{i\}, :\right] = \exp\left(X \cdot W\right) = \exp\left(W\left[\{i\}, :\right]\right)$$

where A a $|\{i\}| \times |V|$ matrix of counts. Observe

$$P\left[i,j\right] = \frac{\exp\left(w_j\right)}{\sum_j \exp\left(w_j\right)}$$

Definition 6. Softmax is a function that converts a vector of real numbers into a probability distribution, i.e., given $v \in \mathbb{R}^n$,

$$\text{softmax}(v) \coloneqq \frac{\exp(v)}{\sum_j \exp(v)}$$

Since we can produce probability distribution vectors, we can also do it for all rows of our weighted matrix to get the same probability matrix as before, i.e.,

$$P[\{i\},:] = \frac{A[\{i\},:]}{\sum_j A[\{i\},:]} = \frac{\exp(W[\{i\},:])}{\sum_j \exp(W[\{i\},:])}$$

as before in order to get the probability distribution of the bigrams that follow.

Remark 2. These matrices A and P can eventually have repeated rows since there are only a finite number of bigrams generated from our dataset.

10 Gradient-Based Learning

When we calculate the NLL using the softmax activation function, i.e.,

$$\begin{aligned} P &= \text{softmax}(X \cdot W) \\ NLL &= -\sum_{(v_i, v_j) \in \{E_k\}_{k=i}^{N}} \log(P[i,j]) \end{aligned}$$

we may initially end up with a loss that is not very good (read, "high"), especially since our weight matrix W is completely randomized. Fortunately, these computations that are differentiable, which means we can calculate a gradient to back-propagate and push the data in a direction that minimizes the loss.

The algorithm to train a neural net for an ngram model is as follows

1. Construct an index matrix XS of ngrams of size $(m, n-1)$ and an output vector ys of size $(m, 1)$
2. Initialize a parameter matrix W of size $((n-1), |V|, |V|)$
3. Until satisfied
   (a) Calculate logits $= W[XS]$
   (b) Calculate $P = \text{softmax}(\text{logits})$
   (c) Calculate loss $= NLL(P)$
   (d) Reinitialize gradient of W to 0
   (e) Back-propagate loss to calculate its gradient
   (f) Calculate $W = W - \text{grad}(\text{loss})$ to reduce the loss

The goal of the neural network described by this process is to modify the initial weights such that the loss observed by the maximum likelihood estimation is very close to one. Our data, assumptions, inputs, and results have not changed, which is good because we can use fundamentally different methods to get the same result.

11 Regularization via learning rate

Training a neural net is a very powerful tool that scales very easily in terms of setup. However, the gradient-based learning is an iterative algorithm that ultimately tries to find a local minima for the loss. If the model tries to construct a non-convex function, then its possible we may update the weights too much and the

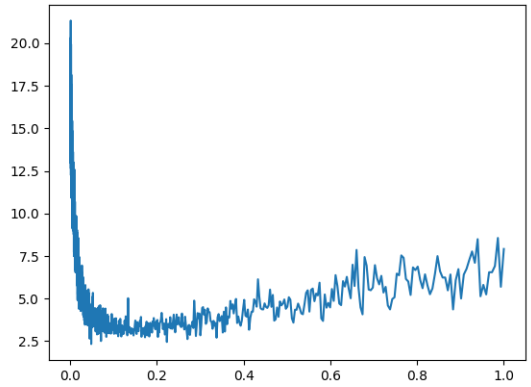


Figure 5: Plot of exponentially stepped learning rates from $[0, 1]$ against the observed loss at each step. Too small of a rate can produce small perturbations in the loss, or slower training, whereas too high of a rate can produce more divergent behavior as the loss bounces around the local minima.

loss can bound back and forth. Fortunately, we can control this using a regularization parameter called the learning rate, denoted by λ, to control the step size, i.e.,

$$W_{i+1} = W_i - \lambda \cdot \text{grad (loss)}$$

So we can choose to take smaller or larger perturbations when updating our weights. See 5 for an example of this applied to our dataset.

11.1 (Initial) Learning rate parameter search

Observe 5 shows a plot the domain against the loss to see what domain λ values results in a lower bound of loss. The highest decreases in loss occur during the smallest rate of change in loss with respect to λ in magnitude, i.e,

$$\left| \frac{d\text{loss}}{d\lambda} - 0 \right| < M$$

for some M. Empirically, this means to calculate the first derivative of $d\text{loss}/d\lambda$ to see where the rate of change decreases to get a sense of where the critical numbers are, i.e., domain subsets that produce maxima and minima.

Remark 3. The function $f(\lambda) = \text{loss}$ being continuous implies $d\text{loss}/d\lambda = 0$ could exist on $\lambda \in \mathbb{R}^+$, but calculating this via an approximation can produce small derivatives that may not be 0 (ideally), but close to 0. We recommend setting M to be the derivative's 5-10th percentile to get λ values that still produce small, steady changes.

11.2 Learning rate decay

Gradient-based learning uses a learning rate as a scale for the gradient. Unfortunately, the stochastic nature of this process implies it is possible for the current loss to bounce back and forth around the local loss minima. Fortunately, this can be mitigated by eventually using a smaller learning rate to make small perturbations back and forth until we "reach" the local loss minima. This technique is called learning rate decay. Assuming you have a good learning rate,

1. Train the model for some time (several iterations)
2. Once you've passed a certain amount of time (or iterations), reduce the learning rate by a factor of 10 or so

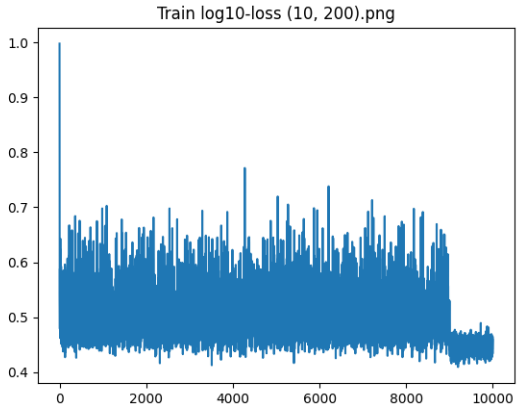


Figure 6: Plot of an extreme example of our neural net's loss when the loss is bounded, but unstable for most of the training loop until we introduce learning rate decay around the 90% epoch.

3. Continue to train until satisfied

Part IV
makemore via Multi-layer Perceptron

12 Stochastic Gradient Descent via Bengio et al. 2003

Recall we calculated the logits, or logarithmic counts using an encoded matrix X and a weight matrix W, and then passed these logits to the softmax activation function to the get the probability distribution for each $v \in V$ and ultimately calculate a loss via NNL, i.e.,

$$\begin{aligned} \text{logits} &= X \cdot W \\ P &= \text{softmax}\left(\text{logits}\right) \\ loss &= NNL\left(P\right) \end{aligned}$$

Definition 7. For a classification neural net, a **forward pass** is a composite function of the calculating the logits, the activation, and the loss.

We can abstract these calculations and even extend them to involve other functions that have their own parameters. The paper Bengio et al. 2003 proposed the following construction of the logits:

$$\text{logits} = \tanh\left(C\left[X\right] \cdot W_1 + b_1\right) \cdot W_2 + b_2$$

where

- X is the tokenized inputs
- C is the $\left(\left|V\right|, k\right)$ mapping that embeds X into a lower-dimensional space
- $W_1$is the (k, hidden) weight matrix
- b_1 is the $(\text{hidden}, 1)$ bias vector

- $W_2$is the (hidden, $|V|$) weight matrix
- b_1 is the ($|V|, 1$) bias vector

In other words, given an embedding transformation C and a linear transformation L_i,

$$\text{logits} = L_2 \circ \tanh \circ L_1 \circ C\,[X]$$

The first linear transformation L_1 that acts on $C\,[X]$ is called the **hidden** layer, and that can be of any size we choose. The tanh activation function is an element-wise function that does not change the dimension of its input. Finally, we need the last linear transformation L_2 in order to change the dimensions of our logits to have $|V|$ columns in order to properly calculate our distribution matrix P.

Our updated stochastic gradient descent algorithm is as follows:

1. Construct an index matrix XS of ngrams of size $(m, n-1)$ and an output vector ys of size $(m, 1)$
2. Initialize the parameters C, W_1, b_1, W_2, b_2.
3. Until satisfied
   (a) Calculate $\text{logits} = L_2 \circ \tanh \circ L_1 \circ C\,[X]$
   (b) Calculate $P = \text{softmax}\,(\text{logits})$
   (c) Calculate $\text{loss} = NLL\,(P)$
   (d) Reinitialize gradient of the parameters
   (e) Back propagate loss to calculate its gradient
   (f) Calculate $W = W - \text{grad}\,(\text{loss})$ to reduce the loss

The goal of the neural network in this exercise is to modify our weights such that our loss observed by the maximum likelihood estimation is very close to the one. Our data, assumptions, inputs, and outputs have not changed, which means we fundamentally are using different methods to get the same result.

13 Machine Learning is all About Representation

Is one model better than another if it has better loss? No. Mind the number of parameters when considering a model's goodness of fit. A neural network's capacity grows with the number of parameters, which also implies the model's capability to overfit increases. This roughly means the model is memorizing your data. Recall the goal of any machine learning model is to construct a means to recreate data from an unknown distribution, and your dataset is but a subset of the true dataset. For example, our dataset is a subset of all possible names, which start with different letters, so the model predicting $< S >$ followed by any one of $\{e, a, o, b, d, etc.\}$ are not incorrect according to the data. An overfit model will only sample from the training data. You do not get new data, and the loss on unseen data could be very high. The solution is to either a) add more data to construct a representative set to train on, or b) fine tune using a train/test/split.

13.1 Train/Test Split

The training set is for optimize the model's parameters using gradient descent. The validation set is to optimize the hyper-parameters, like learning rate and hidden layer sizes, embedding sizes, etc. The test set should only be used to evaluate model quality, and used **very** sparingly

If the training and validation loss are roughly the same? The model is under-fitting, so the model is tiny, which means we can gain improvements by sizing up.

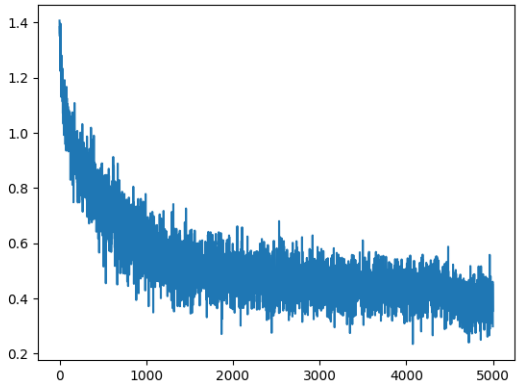


Figure 7: The classic hockey-stick graph that is shown in many machine learning tutorials. Observe the first 1000 iterations are just squashing the gradients to match the initial problem, and then the true training happens afterwards.

14 Activations & Gradients

14.1 Initial Loss

Observe the loss in the initial iterations of the training loop starts off very high then rapidly decreases – depicting that classic hockey-stick graph in 7. This is a sign the neural network's weights are improperly initialized. Remember, neural networks are a representation and approximation problem. One should have a rough idea for what loss to expect at initialization depending on the loss function and the problem description. The neural net for our classification problem predicts the next token $v \in V$ given a token sequence $\{v\}_{v \in V}$, then the worse case scenario implies we have no distinguishing information about our token set, i.e., $v \in V$ are uniformly likely. Therefore, the principle of indifference implies

$$P(v|X) = \frac{1}{|V|}$$

and then the negative log-likelihood loss would be $NNL(P(v|X))$. For $|V|$ consisting of 26 tokens and 1 stop token, then

$$NNL\left(\frac{1}{|V|}\right) \approx 3.29$$

If our initial loss is much higher than we expect, then means the neural net initially carries the assumption X that some $v \in V$ are more important than others, so the loss calculated against these incorrect probability distributions will be very high. Eventually, the gradient will push the weights in the correct direction, but an incorrect initialization results in a waste of computational time and power.

Example 7. Let $|V| = 3$ so we only have to predict one of three labels. The softmax probabilities calculated from improperly initialized logits can result in high loss as demonstrated in 1. Observe a massive difference in magnitude between two floating point numbers that are passed to the softmax can actually lead to a severe rounding error, resulting in what appears to be zero loss.

The solution is to initialize the weights to very small numbers to ensure there is some entropy, which is useful for symmetry breaking. See 8 for the effect.

Remark 4. One could initialize the weights to all zeros, but the consequence is the gradients associated with those weights could stay zero, which is very bad.

Description	Logits	$P = \text{softmax}(X)$	Loss
Zeros	$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0.3333 & 0.3333 & 0.3333 \end{bmatrix}$	1.0986
$U([0,1])$	$\begin{bmatrix} 0.4946 & 0.1297 & 0.2467 \end{bmatrix}$	$\begin{bmatrix} 0.4041 & 0.2806 & 0.3154 \end{bmatrix}$	1.1540
High probability of label, low loss	$\begin{bmatrix} 0 & 0 & 5 \end{bmatrix}$	$\begin{bmatrix} 0.0066 & 0.0066 & 0.9867 \end{bmatrix}$	0.0133
Low probability of label, high loss	$\begin{bmatrix} 0 & 5 & 0 \end{bmatrix}$	$\begin{bmatrix} 0.0066 & 0.9867 & 0.0066 \end{bmatrix}$	5.0133
$N(0,1)$, not great	$\begin{bmatrix} 1.3660 & 1.0109 & 0.2483 \end{bmatrix}$	$\begin{bmatrix} 0.4931 & 0.3457 & 0.1612 \end{bmatrix}$	1.8248
$N(0,10^2)$, high loss	$\begin{bmatrix} 11.9580 & -10.3641 & 9.7887 \end{bmatrix}$	$\begin{bmatrix} 8.9746E^{-1} & 1.8140E^{-10} & 1.0254E^{-1} \end{bmatrix}$	2.2775
$N(0,100^2)$, higher loss	$\begin{bmatrix} -212.3578 & -144.7072 & 6.0214 \end{bmatrix}$	$\begin{bmatrix} 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$	0.0

Table 1: Examples of Calculated Losses Against Different Logits

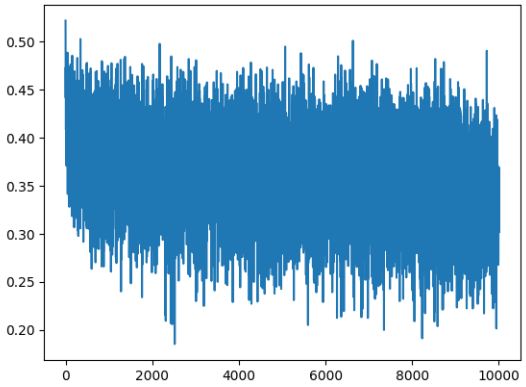


Figure 8: The same model with properly initialized weights. Observe the loss is bounded by a much smaller range.

14.2 Saturated tanh (Hyperbolic Tangent)

Our neural network is an unknown composite function that calculates a single number called the loss. We use non-linear functions called **activation functions** in our function in order to capture the non-linearity of the data. Recall the derivative at a point describes the rate of change at that point, or slope. If the derivative is positive or negative, the function at that point increases or decreases respectively. The **stochastic gradient descent algorithm** (SGD) is an optimization algorithm that calculates the partial derivatives (or gradients) of this unknown composite function's inputs, and uses these derivatives to make small changes to the inputs in order to minimize its loss output. If these derivatives are zero, then the function is a constant, which means we cannot affect the inputs.

The activation we used by following Bengio et al. 2003 is $\tanh(x)$ (hyperbolic tangent). This function has a domain of $x \in \mathbb{R}$ and a range of $\tanh(x) \in (-1, 1)$, which is why its called a "squashing function". Recall the derivative of our activation function is

$$\tanh'(x) = 1 - (\tanh(x))^2$$

Solving for $\tanh'(x) = 0$ yields $\tanh(x) = \pm 1$. Observe this derivative is zero when $\tanh(x)$ is along the boundary of its range. Mathematically speaking, this only occurs when we take

$$\lim_{x \to \pm\infty} \tanh(x) = \pm 1$$

How soon does this limit occur? Let's solve for it analytically. Observe for some $M \in \mathbb{R}$,

$$\begin{aligned}
\tanh(x) =& \frac{e^x - e^{-x}}{e^x + e^{-x}} = M \\
\implies & e^x - e^{-x} = M\left(e^x + e^{-x}\right) \\
\implies & e^{2x} - 1 = M\left(e^{2x} + 1\right) \\
\implies & e^{2x}(1 - M) = 1 + M \\
\implies & e^{2x} = \frac{1+M}{1-M} \\
\implies & x = \frac{\ln\left(\frac{1+M}{1-M}\right)}{2}
\end{aligned}$$

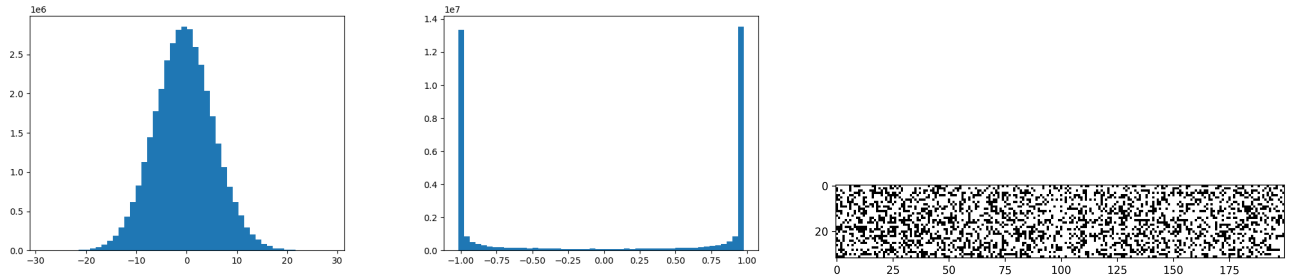
For $M = 0.99$, then $x \approx 2.64$. Indeed, this phenomena occurs as early as $|x| \leq 3$,

$$\begin{aligned}
\tanh(3) &\approx 0.995 \\
\tanh(-3) &\approx -0.995 \\
\tanh(5) &\approx 0.9999 \\
\tanh(-5) &\approx s - 0.9999
\end{aligned}$$

This is dangerous because floating point approximation on modern computers implies rounding errors can accumulate and we can get close to zero gradients without actually taking a limit! The consequence of this is our variance decreases. This phenomena in machine learning context is referred to as **saturation**. See figure 9 for a demonstration of how a saturated domain would result in a sparse gradient map whose learning potential is really limited since most of the gradients are zeroed out. In contrast, figure 10 shows an unsaturated domain resulting in a dense gradient map such that every neuron has the ability to update. Saturation is problematic because it kills the gradient, which potentially introduces "dead" neurons that may never be updated since the chain rule could always result in zero.

Example 8. Assume we take a small training set sample X with very saturated pre-activation values

$$\begin{aligned}
M &= L_1 \circ C[X] \\
M[i,j] &= A \cdot C[x] + b
\end{aligned}$$

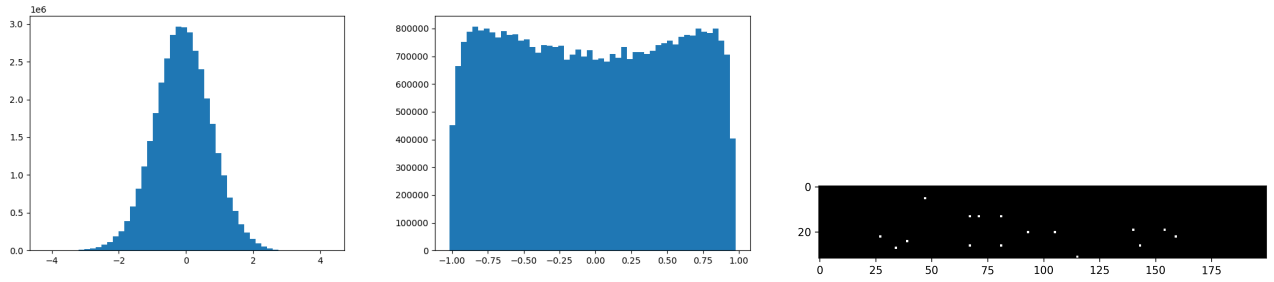


(a) Pre-activation values largely outside $\tanh(x)$ range

(b) A very active hidden layer

(c) Destroyed gradient map (large white tile count)

Figure 9: A saturated activation layer will effectively kill the gradients



(a) Pre-activation values densely inside $\tanh(x)$ range

(b) A very inactive hidden layer

(c) Dense gradient map (small white tile count)

Figure 10: An unsaturated activation layer resulting in a good gradient map

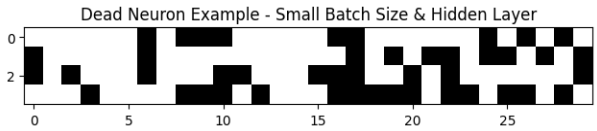


Figure 11: Dead neuron resulting from a small batch size (all white column)

such that $2.64 < M[i,j]$ for all i and a fixed j. Thus, activation $\tanh(M[i,j]) \in (0.9, 1)$. The gradient of the activation is

$$\begin{aligned}
\frac{\partial \tanh(M[i,j])}{\partial x} &= \frac{\partial \tanh(A \cdot C[x] + b)}{\partial x} \\
&= \left(1 - (\tanh(A \cdot C[x] + b))^2\right) \cdot \frac{\partial (A \cdot C[x] + b)}{\partial x} \\
&= \left(1 - (\tanh(A \cdot C[x] + b))^2\right) \cdot A \cdot C[x] \\
&= (0) \cdot A \cdot C[x] \\
&= 0
\end{aligned}$$

for all i. This is a very bad situation since $M[:,j]$ cannot easily be updated via

$$M[:,j] = M[:,j] - \text{grad}(\tanh(M[:,j]))$$

The column j of the pre-activations is considered a "dead" neuron since its gradients are all zero, so the inputs are going to stay constant, implying they cannot be updated via the back-propagation algorithm. This is visualized in figure 11.

14.3 Kaiming Unit

Proposition 6. *Let X be a standard normal random variable, i.e., $X \sim \mathcal{N}(0,1)$. The expectation of $\tanh^2(X)$, i.e.,*

$$E\left[\tanh^2(X)\right] = \int_{-\infty}^{\infty} \tanh^2(x) \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx \approx 0.39$$

Remark 5. This integral has no closed form solution, but it is less than 1 since

$$\tanh(x)^2 = \frac{sinh^2(x)}{cosh^2(x)}$$

and someone claimed $sinh(x) < cosh(x)$ **TODO: ON WHAT INTERVAL?**

Proof. We'll use integration by parts.

Let

$$\begin{aligned}
u = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) v = x - \tanh(x) \\
du = -x\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx dv = \tanh^2(x)\, dx
\end{aligned}$$

Then

$$\int_{-\infty}^{\infty} \tanh^2(x) \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx = uv - \int v du$$

$$= \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)(x - \tanh(x))\Big|_{-\infty}^{\infty} + \int_{-\infty}^{\infty} x \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)(x - \tanh(x))\, dx$$

$$= \frac{1}{\sqrt{2\pi}}(0)\left(\left(\lim_{x\to\infty} x - 1\right) - \left(\lim_{x\to-\infty} x + 1\right)\right) + \int_{-\infty}^{\infty} x \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)(x - \tanh(x))\, dx$$

$$= \int_{-\infty}^{\infty} x^2 \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx - \int_{-\infty}^{\infty} x \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \tanh(x)\, dx$$

Observe since $X \sim N(0,1)$,

$$\int_{-\infty}^{\infty} x^2 \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx = E\left[X^2\right] = Var[X] - E[X]^2$$

so the first integral evaluates to 1.

For the second integral, let $u = x^2$, then $du = 2xdx$ and substitution yields

$$\int_{-\infty}^{\infty} x \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \tanh(x)\, dx = \frac{1}{2}\int_{0}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{u}{2}\right) \tanh\left(\sqrt{u}\right) du$$

$$= \frac{1}{2}\frac{1}{\sqrt{2\pi}}\int_{0}^{\infty} \exp\left(-\frac{1}{2}u\right) \tanh\left(\sqrt{u}\right) du$$

$$= \frac{1}{2\sqrt{2\pi}} \mathcal{L}\left[\tanh\left(\sqrt{u}\right)\right]_{s=\frac{1}{2}}$$

where $\mathcal{L}$ is the Laplace Transform.

Hence,

$$E\left[\tanh^2(X)\right] = 1 - \frac{1}{2\sqrt{2\pi}} \mathcal{L}\left[\tanh(x)\right]_{s=\frac{1}{2}}$$

since the latter has no closed form. □

Definition 8. Let Z be a standard normal random variable and f be an activation function. The gain is

$$\frac{1}{\sqrt{E\left[f(Z)^2\right]}}$$

Corollary 1. *The gain of activation function* $\tanh(x)$ *is 1*

Proof. By 6,

$$\frac{1}{\sqrt{E\left[\tanh^2(Z)\right]}} \approx \frac{1}{\sqrt{0.39}} \approx 6.57$$

□

Proposition 7. *Let X be a standard normal random variable. The expectation of* $\tanh(X)$*, i.e.,*

$$E[\tanh(X)] = \int_{-\infty}^{\infty} \tanh(x) \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx$$

Proof. We'll use integration by parts.

Let

$$u = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) v = \log(cosh(x))$$
$$du = -x\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx dv = \tanh(x)\, dx$$

Then

$$\begin{aligned}
\int_{-\infty}^{\infty} \tanh^2(x) \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx &= uv - \int v du \\
&= \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \log(cosh(x))\Big|_{-\infty}^{\infty} + \int_{-\infty}^{\infty} x\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \log(cosh(x))dx \\
&= \frac{1}{\sqrt{2\pi}} (0) \left(\lim_{x\to\infty} \log(cosh(x)) - \lim_{x\to-\infty} \log(cosh(x))\right) + \int_{-\infty}^{\infty} x\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \log(cosh(x \\
&= \frac{1}{\sqrt{2\pi}} (0)(0) + \int_{-\infty}^{\infty} x\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \log(cosh(x))dx \\
&= \int_{-\infty}^{\infty} x\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \log(cosh(x))dx
\end{aligned}$$

Let $u = x^2$, then $du = 2xdx$ and substitution yields

$$\begin{aligned}
\int_{-\infty}^{\infty} x\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \tanh(x)\, dx &= \frac{1}{2}\int_{0}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{u}{2}\right) \log\left(cosh\left(\sqrt{u}\right)\right) du \\
&= \frac{1}{2\sqrt{2\pi}} \mathcal{L}\left[\log\left(cosh\left(\sqrt{u}\right)\right)\right]_{s=\frac{1}{2}}
\end{aligned}$$

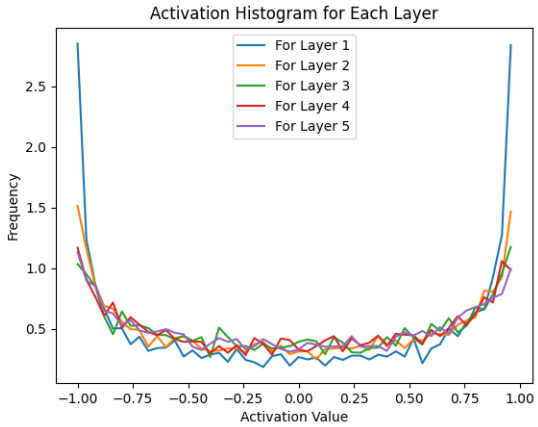
where $\mathcal{L}$ is the Laplace Transform.

Hence,

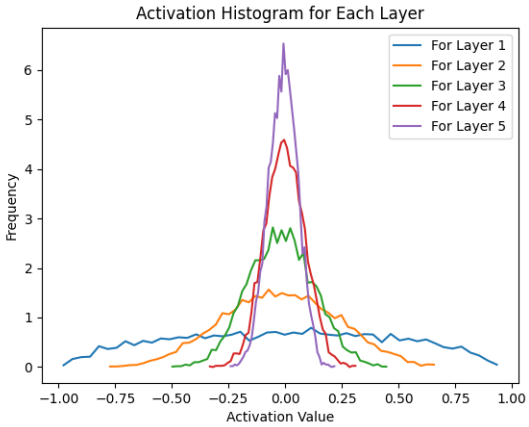
$$E[\tanh(X)] = 1 - \frac{1}{2\sqrt{2\pi}} \mathcal{L}\left[\log(cosh(x))\right]_{s=\frac{1}{2}}$$

since the latter has no closed form. □

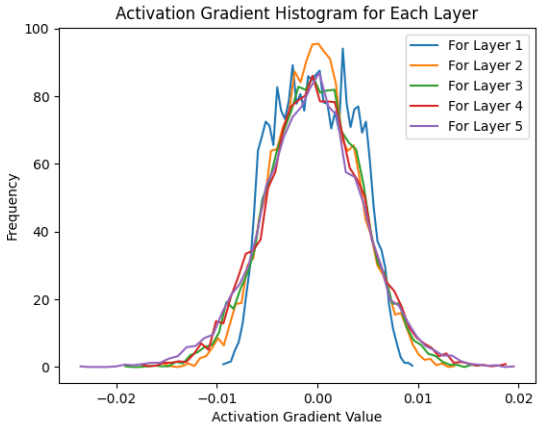
Var(tanh(x))~=0.42~= 1/gain^2
- In tanh(x) the input value is range between -1 to 1 and
- In tanh(x)^2 the input range is between 0 to 1.
- the PDF function of Gaussian distribution is exp(-x^2/2)/sqrt(2*pi)
the output of these function is 0.39. (variance)
Std = sqrt(0.39) = 5/3. <———— WRONG
so we multiply this value for conserve the std of the distribution.



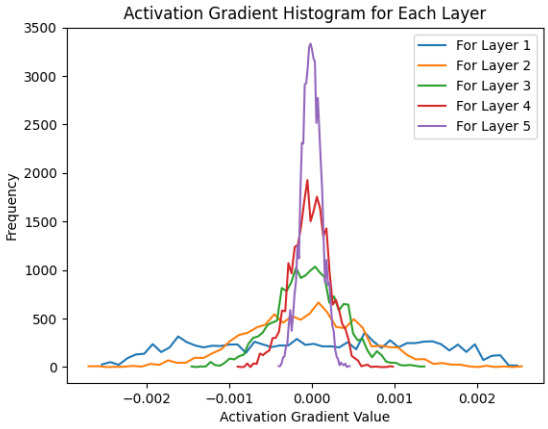
(a) Properly Initialized Activations



(b) Improperly Initialized Activations



(c) Properly Initialized Activation Gradients



(d) Improperly Initialized Activation Gradients

Figure 12: Distributions of activations and activation gradients in the activation layers of an MLP with tanh activations.

The histograms of the forward pass activations for each tanh layer in 12 allow us to gauge how often they lie outside the $(-1, 1)$ range, since the neurons associated with those values will have greatly reduced gradients and consequently reduced learning potential. The properly initialized activations (and gradients) are stay within a consistent range no matter the layer depth, whereas the improperly initialized activations (and gradients) continue to shrink with each layer. Observe how the activation (and gradient) values are within a small window centered around 0, with the properly initialized ones being mostly near the boundary of these windows and the improperly initialized ones slowly converging to 0 as the number of layers increases.

15 Batch-Normalization

The pre-activation states for tanh need to be roughly distributed as a standard normal (Gaussian) in order to ensure we do not get layers of sparse gradients. The **batch-normalization** process roughly states the pre-activation states should be normalized to follow a Gaussian distribution before passing them over to the activation function. In other words, given pre-activation H,

$$BN\left(H, \gamma, \beta\right) = \gamma \frac{\left(H - E\left[H\right]\right)}{\sqrt{Var\left[H\right] + \varepsilon}} + \beta,$$

in which γ (gain) and β (bias) are shifting terms, and ε is a constant added for numerical stability. Observe this operation is differentiable. In practice, this normalization is commonly performed after linear and convolutional layers. We use batch-normalization because it is quite effective at controlling the activations and their distributions.

Batch-normalization introduces stability for training because it effectively controls the inputs for activation functions and their respective distributions, but this process is prone to errors because it's "unnatural" that randomly picked samples have a mathematical impact on what those samples actually represent. In practice, one would prefer the neural network to adjust its weights to be more diffuse, sharper, have trigger-happy gradients, have non-trigger-happy gradients (biased?), i.e., allow back-propagation to shift the weight's distribution in the proper direction. That is why we need the gain and bias terms to also have adjustable gradients so that we can scale the neurons to fit the neural network's needs.

Proposition 8. *Let L be a linear transformation and X be an input. The bias term in a linear layer passed to batch-normalization will have zero gradient, i.e.,*

$$BN\left(L\left(X\right), \gamma, \beta\right) = \gamma \frac{W\left(X - E\left[X\right]\right)}{\sqrt{W^2 Var\left[X\right]}} + \beta$$

for some γ, β.

Proof. Recall

$$L\left(X\right) = WX + b$$

for some weight matrix W and bias vector b. Observe this bias vector cancels out in the numerator when performing batch-normalization since expectation is linear, i.e.,

$$\begin{aligned} L\left(X\right) - E\left[L\left(X\right)\right] &= WX + b - E\left[WX + b\right] \\ &= W\left(X - E\left[X\right]\right) \end{aligned}$$

Also, recall the variance ignores constants, so

$$Var\left[L\left(X\right)\right] = Var\left[WX + b\right] = WVar\left[X\right]W^T$$

Thus,

$$BN\left(L\left(X\right), \gamma, \beta\right) = \gamma \frac{W\left(X - E\left[X\right]\right)}{\sqrt{WVar\left[X\right]W^T}} + \beta$$

□

This implies we can safely ignore the bias from the linear layer in any implementations in favor of the batch-normalization's own bias.

We make the following adjustment to our stochastic gradient descent algorithm to implement batch-normalization:

1. $\text{logits} = L_2 \circ \tanh \circ B$

Infinite Impulse Response (IIR) filter

15.1 Reproducibility Cost via Noise Introduction

The stability offered by batch-normalization actually comes at a terrible cost. We currently calculate logits via a deterministic process that passes a single example feeding into a neural net via

$$\text{logits}(x) = L_2 \circ \tanh \circ L_1 \circ x,$$

in which x can be a single input. We can extend this to feed batches of inputs for efficiency reasons, i.e.,

$$\text{logits}(X) = L_2 \circ \tanh \circ L_1 \circ X$$

and these batches are processed independently. However, batch-normalization calculates the logits via sampling,

$$\text{logits}\left(X\right) = BN\left(*, \gamma, \beta\right) \circ L_2 \circ \tanh \circ BN\left(*, \gamma, \beta\right) \circ L_1 \circ X$$

which is prone to domain issues as we can no longer use a single input and reproducibility problems depending the batch size and the samples themselves. Normalizing the hidden states to be Gaussian implies the statistics of the mean and the standard deviation are going to be impacted because there is a "jitter/noise" accumulated through the calculations. Fortunately, this turns out to be helpful in neural network training because this noise pads out the inputs and introduces small amounts of entropy that help prevent the neural network from overfitting to specific data pieces. Batch-normalization acts as a regularizer to this effect and is subject to strange results and lots of bugs. This phenomena is an example of **data augmentation**. Other normalization techniques have been introduced to compensate for these problems, such as LER normalization, instance normalization, and group normalization.

Machine learning models are useful, but they have to be deployed to make them useful, e.g., accessed in applications like classifiers or predictors. We would like to be able to feed in a single individual example and get a prediction from our neural net trained via batch-normalization. How can we do this given the network structure has been altered to only accept batches? The trick is to setup the neural network to calculate the mean and standard deviation in a running manner during training.

15.2 As a train-only operation

After training a neural network that uses batch-normalization, we can use the batch-normalizations's calculated parameters to construct an explicit expression that uses the weights of the preceding Linear layers, effectively erasing the need to perform the batch-normalization calculations at inference time.

Let H be result of a linear hidden layer calculation, i.e.,

$$H = L\left(X\right) = WX + b$$

for some weigh W and bias b. If we apply batch-normalization to H, then

$$BN\left(H, \gamma, \beta\right) = \gamma \frac{\left(H - E\left[H\right]\right)}{\sqrt{Var\left[H\right] + \varepsilon}} + \beta$$

Let W_t and b_t be the resultant weights and bias terms of a trained linear network, and let σ^2 and μ be the batch-normalization sample variance and mean, so that we can get the explicit composition function

$$BN\left(H, \gamma, \beta\right) = BN\left(L\left(X\right), \gamma, \beta\right)$$

Observe

$$\begin{aligned} BN\left(L\left(X\right),\gamma,\beta\right) &= \gamma\frac{\left(W_tX+b_t-\mu\right)}{\sqrt{\sigma^2+\varepsilon}}+\beta \\ &= \gamma\frac{\left(W_tX\right)}{\sqrt{\sigma^2+\varepsilon}}+\gamma\frac{\left(b_t-\mu\right)}{\sqrt{\sigma^2+\varepsilon}}+\beta \\ &= W_t\frac{\gamma}{\sqrt{\sigma^2+\varepsilon}}X+\frac{\gamma\left(b_t-\mu\right)}{\sqrt{\sigma^2+\varepsilon}}+\beta \\ &= \left(W_t\frac{\gamma}{\sqrt{\sigma^2+\varepsilon}}\right)X+\left(\frac{\gamma\left(b_t-\mu\right)}{\sqrt{\sigma^2+\varepsilon}}+\beta\right) \end{aligned}$$

Let

$$\begin{aligned} W_{new} &\coloneqq \left(W_t\frac{\gamma}{\sqrt{\sigma^2+\varepsilon}}\right), \\ b_{new} &\coloneqq \left(\frac{\gamma\left(b_t-\mu\right)}{\sqrt{\sigma^2+\varepsilon}}+\beta\right) \end{aligned}$$

so that the resultant computation is explicitly batch-normalization,

$$BN\left(X,\gamma,\beta\right) = W_{new}X+b_{new}$$

The advantage of this is we get the same forward pass during inference, i.e., we see that the batchnorm is there just for stabilizing the training, and can be thrown out after training is done!

Part V
Language Models

ChatGPT (or Chat Generative Pre-trained Transformer) has taken the world by storm. It is a probabilistic language model that spits out chunks of text when given a prompt. A language model takes in a sequence of tokens and completes the sequence. Recall a sequence is composed of elements from a set called a vocabulary. To **tokenize** is to convert raw inputs to a sequence according to some vocabulary of possible elements. For example, tokenizing an input for a character-level language model can result in a sequence of indices of characters in the ordered vocabulary, i.e., if $V = \{a, b, d, < S >\}$ with 0-index set $\{0, 1, 2, 3\}$, an input "bad dab" can be represented as $(1, 0, 2, 3, 2, 0, 1)$.

A sequence recursively contains examples of other sequences since its elements follow one another, so training on a single sequence also simultaneously trains the model to make predictions at every position in the sequence. For example, $(1, 0, 2, 3, 2, 0, 1)$ contains contexts of sizes 1 through 6 for character predictions, e.g.,

$$\begin{aligned} 1 &\to 0 \\ (1,0) &\to 2 \\ (1,0,2) &\to 3 \\ (1,0,2,3) &\to 2 \\ (1,0,2,3,2) &\to 0 \\ (1,0,2,3,2,0) &\to 1 \end{aligned}$$

This allows the transformer to get used to seeing contexts of various sizes as well. This implies the transformer during inference will be able to complete sequences that start with just a single token. It is also computationally expensive and prohibitive to pass in all the text at once for training, so we recommend setting a fixed context size. This also means the transformer trained on a fixed block size will be unable to predict sequences that are larger than the block size.

16 Attention is All You Need

Each embedding has a batch, time, and channel dimension. The time dimension represents the sequence length, which can considered the context window size the model considers when making predictions. The channel dimension represents the features or depth of the data at each time step, e.g., some pictures have three channels called RGB, and text applications equate the channels with the embedding size.

The tokens should be coupled in a specific way that can only look behind, not ahead, i.e., information only flows from previous context to the current time step. We cannot use future tokens as that would lead to a data leak, i.e., be cheating, which is bad because then any accuracy metrics would be misleading.

Example 9. The simplest way to for the current token to communicate with its previous context is to take an average across the channels. This communication is pretty weak since its very lossy, and does not capture any spatial arrangements. In machine learning for text applications, this is known as Bag of Words (BOW).

Given a batch tensor $X \in M_{T\times C}(\mathbb{R})$ for sequence length T, embedding size C, the bag of words is

$$BOW(X,t) = \frac{1}{t}\sum_{i=1}^{t} X[i,:]$$

which is the average of all time steps up and including t.

16.1 Efficient Bag of Words (BOW)

The bag of words we described is a very inefficient operation as we would have to iterate over every batch in order to construct the output matrix row-wise. Fortunately, there is a trick.

Proposition 9. *Let X the batch of $T\times C$, where T denotes the sequence length, and C denotes the embedding dimension. Let $\overline{L}$ be a $T\times T$ lower triangular matrix of ones that has been normalized by row-sums. Then the Bag of Words matrix for a single batch can be represented as*

$$\{BOW(X,t)\}_{t=1}^{T} = \overline{L}\cdot X$$

Proof. Observe if we ignore the normalizing scalars,

$$\{t\cdot BOW(X,t)\}_{t=1}^{3} = \left[\sum_{i=1}^{t} X[i,:]\right]_{t=1}^{T} = \begin{bmatrix} \sum_{i=1}^{1} X[i,:] \\ \sum_{i=1}^{2} X[i,:] \\ \sum_{i=1}^{3} X[i,:] \\ \vdots \\ \sum_{i=1}^{T} X[i,:] \end{bmatrix} = \begin{bmatrix} X[1,:] \\ X[1,:]+X[2,:] \\ X[1,:]+X[2,:]+X[3,:] \\ \vdots \\ X[1,:]+X[2,:]+X[3,:]+\cdots+X[T,:] \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ \vdots & \vdots & \vdots \\ 1 & 1 & 1 \end{bmatrix}$$

where L is a $T\times T$ lower-triangular matrix of ones, i.e.,

$$L_{ij} = \begin{cases} 1 & j \le i \\ 0 & i < j \end{cases}.$$

Now, to reproduce the normalizing scalars, we need to divide by the row-sums of L, i.e.,

$$\overline{L}_{ij} = \begin{cases} \frac{1}{i} & j \le i \\ 0 & i < j \end{cases}.$$

Then

$$\overline{L}\cdot X = \begin{bmatrix} \frac{1}{1}\sum_{i=1}^{1} X[i,:] \\ \frac{1}{2}\sum_{i=1}^{2} X[i,:] \\ \frac{1}{3}\sum_{i=1}^{3} X[i,:] \end{bmatrix} = \{BOW(X,t)\}_{t=1}^{3}$$

as desired. □

16.2 The Wei Matrix

Definition 9. The affinity between tokens is the pair-wise probability distribution, i.e., $P\left(v|\left(\prod_{i=1}^{T-1} v_i\right)X\right)$.

The non-zero elements of $\overline{L}$ follow a uniform distribution row-wise, i.e., have uniform affinity. This is a problem we will need to solve, but first, we can deconstruct the $\overline{L}$ matrix matrix described in 9 by recognizing it as the output of a softmax function.

Proposition 10. *Let $\overline{L}$ be a $T \times T$ lower triangular matrix of ones that has been normalized by row-sums. Then there exists a $T \times T$ matrix W such that*

$$\overline{L} = \lim_{x \to -\infty} softmax\left(W\right)$$

Explicitly, W is an upper-triangular matrix with a zeroed diagonal and $-\infty$ for all elements, i.e.,

$$W = \begin{cases} 0 & j \leq i \\ -\infty & i < j \end{cases}$$

In the literature, this W is known as a **wei** matrix, which contextually contains normalized affinity scores.

Proof. Recall $softmax$ uses the exponential function, which has the following identities,

$$\begin{aligned} \exp\left(0\right) &= 1 \\ \lim_{x \to -\infty} \exp\left(x\right) &= 0 \end{aligned}$$

so

$$L_{ij} = \begin{cases} 1 & j \leq i \\ 0 & i < j \end{cases} = \begin{cases} \exp\left(0\right) & j \leq i \\ \lim_{x \to -\infty} \exp\left(x\right) & i < j \end{cases}$$

Define W to be the $T \times T$ upper triangular matrix,

$$W \coloneqq \begin{cases} 0 & j \leq i \\ \lim_{x \to -\infty} x & i < j \end{cases}$$

so we have$L_{ij} = \exp\left(W\right)$.

For $T = 3$, this looks like

$$L_{3\times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \lim_{x \to -\infty} \exp \begin{bmatrix} 0 & x & x \\ 0 & 0 & x \\ 0 & 0 & 0 \end{bmatrix} = \lim_{x \to -\infty} \exp\left(W\right)$$

Then

$$\overline{L} = \frac{L_{ij}}{\sum_k L_{ik}} = \lim_{x \to -\infty} \frac{\exp\left(W_{ij}\right)}{\sum_k \exp\left(W_{ik}\right)} = \lim_{x \to -\infty} softmax\left(W\right).$$

□

Remark 6. This expression is well-defined but moot since most of our work is going to be performed in the computer space, but well-written software that can perform mathematical computations like C++ or python can represent $-\infty$, so we won't need the limit expression there.

To summarize, we can perform weighted aggregations of past elements in a given context via matrix multiplication of carefully constructed triangular matrices, first via $\overline{L}$ and then via W. These $T \times T$ matrices track the affinities between different tokens in a row-wise manner in their lower triangular entries, as each row represents the recurrent context sizes and the their futures. They also capture each recurrent context's scope. By design, the zero elements in our $W\left[i,:\right]$ represent the past and current context tokens, whereas the $-\infty$ elements represent the future. We do this to ensure the future cannot communicate with the past.

We will soon see later in attention blocks that these wei matrices are application dependent, and do not have to be triangular.

For NLP and autoregressive applications, a wei matrix W with only zeros and infinities implies all of the affinities between the tokens stored in $\overline{L}$ will follow the uniform distribution, which is a problem because we want them to be data dependent, e.g., the position of a vowel character in a given context should and desire to look for associated consonants in its past as they will be able to form comprehensible language represented in the dataset. We will see how attention solves this problem by making our wei matrix construction data dependent.

17 Attention Blocks

Let X be an a single batch input matrix of size (T, C) containing the embedded token and position information.

Definition 10. A linear projection on an input matrix X is the multiplication

$$XW$$

where W is a (C, d) matrix for some d.

Definition 11. Given batch input matrices X_i of size $T \times C$, and given linear maps W_Q, W_K, W_V of size (T, d), an **attention block** is the product of linear projections in the following way,

$$Attention \coloneqq softmax\left(\frac{(X_1W_Q)(X_2W_K)^T}{\sqrt{d}}\right)(X_3W_V)$$

Let's calculate the final dimensions of attention. Since each $X_iW_i \in (T, d)$

$$\begin{aligned}\dim\left((X_iW_Q)(X_iW_K)^T(X_3W_V)\right) &= (T,d)\cdot(T,d)^T\cdot(T,d)\\ &= (T,T)\cdot(T,d)\\ &= (T,d)\end{aligned}$$

This dimension d is known as the **head-size**, and is useful as a hyper-parameter. In the literature, X_iW_Q is known as the **query** space, XW_K is known as the **key** space, and XW_V is known as the **value** space.

We scale by the square root of d since unit-Gaussian variants when multiplied together will reach a magnitude close to d.

Proposition 11. *Let linear transformations $X_iW_j \sim \mathcal{N}(0,1)$ be of size $T \times d$. Then*

$$Var\left(\frac{(X_iW_Q)\cdot(X_iW_K)^T}{\sqrt{d}}\right) = 1$$

Proof. By assumption, each row $(X_iW_Q)[i,:]$ and $(X_iW_K)[i,:]$ follows $\mathcal{N}(0,1)$, then

$$\begin{aligned}Var\left((X_iW_Q)[i,:]\cdot(X_iW_K)[i,:]\right) &= Var\left((X_iW_Q)[i,:]\right)\cdot Var\left((X_iW_K)[i,:]\right)\\ &= 1\end{aligned}$$

Since (X_iW_Q) and (X_iW_K) are size $T \times d$, then the total variance is

$$\begin{aligned}Var\left((X_iW_Q)\cdot(X_iW_K)^T\right) &= \sum_{i=1}^{d} Var\left((X_iW_Q)[i,:]\right)\cdot Var\left((X_iW_K)[i,:]\right)\\ &= \sum_{i=1}^{d} 1\\ &= d\end{aligned}$$

Finally, recall $Var(aX) = a^2 Var(X)$, thus

$$Var\left(\frac{(X_i W_Q) \cdot (X_i W_K)^T}{\sqrt{d}}\right) = \frac{d}{\left(\sqrt{d}\right)^2} = \frac{d}{d} = 1$$

□

This fact is useful for also dispersing values, since extreme values passed to $softmax$ can also result in extreme probabilities and possibly dead gradients. This can be described as $softmax$ approaching the largest value in the vector, which can visually interpreted as sharp peaks when plotted.

Definition 12. An attention block is **self-attentive** if all X_i are the same, i.e.,

$$SelfAttention \coloneqq softmax\left(\frac{(XW_Q)(XW_K)^T}{\sqrt{d}}\right)(XW_V),$$

then we can write the attention block more compactly, i.e.,

$$Attention \coloneqq softmax\left(\frac{QK^T}{\sqrt{d}}\right)V$$

Definition 13. An attention block is **cross-attentive** if the X_i passed to the W_Q and W_K transformations do not equal the X_j passed to the W_V transformation for some j.

Example 10. In practice, the $(XW_Q)(XW_K)^T$ is converted to a wei matrix by taking an upper triangular mask before passing it to softmax so we don't introduce any context leakage in autoregressive models (models that predict future events based on past ones) like time-series or NLP applications, i.e.,

$$Attention = softmax\left(triu\left(\frac{(XW_Q)(XW_K)^T}{\sqrt{d}}, -\infty\right)\right)(XW_V) = softmax(W)V$$

For token prediction applications, this W is the same derived in 10.

Definition 14. An attention block is called an **encoder** if the $softmax$ function input contains the entire context, i.e., if the $(XW_Q)(XW_K)^T$ transformation is not passed through an upper triangular mask.

Definition 15. An attention block is called a **decoder** if the $softmax$ function input contains only the past and present context, i.e., if the $(XW_Q)(XW_K)^T$ transformation is passed through an upper triangular mask.

Attention is a communication mechanism that works on directed graph structures, whose goal is to aggregate information via a weighted sum in an independent manner. For our character prediction model, each sequence of tokens $v_1 \cdots v_n$ can be arranged in a directed graph in which $v_i \to v_{i+1}$ and the last token, v_n, satisfies $v_n \to v_n$. Attention also has no sense of space, and simply acts over a set of vertices in a graph. These vertices have no understanding of their spatial positions, and so it must be provided in order to anchor the vertices to specific positions. This differs from convolution, which has filters that require its information inputs to have a specific layout.

17.1 Multi-Headed Attention

Let i be the ith attention block and W^Q, W^K, W^V represent the query, key, and value linear transformations. The multi-headed attention block can be expressed as

$$\begin{aligned} MultiHead\left(\left\{X_i^1, X_i^2, X_i^3\right\}, \left\{W_i^Q\right\}, \left\{W_i^K\right\}, \left\{W_i^V\right\}\right) &= \bigoplus_{i=1}^{N} Attention\left(\left\{X_i^1, X_i^2, X_i^3\right\}, W_i^Q, W_i^K, W_i^V\right) \\ &= \bigoplus_{i=1}^{N} softmax\left(\frac{\left(X_i^1 W_i^Q\right)\left(X_i^2 W_i^K\right)^T}{\sqrt{d}}\right)\left(X_i^3 W_i^V\right) \end{aligned}$$

Again, if all X_i^j are the same for all j, then this simplifies to the multi-headed self-attention block, i.e.,

$$MultiHead\left(\{X_i\}, \left\{W_i^Q\right\}, \{W_i^K\}, \{W_i^V\}\right) = \bigoplus_{i=1}^{N} Attention\left(X_i, W_i^Q, W_i^K, W_i^V\right)$$

Multi-headed attention applies multiple attentions in parallel and then concatenates the results across the communication channels (heads) to form a single tensor of the embedding dimension. In practice, would set each head size to be the embedding dimension divided by the number of attention heads. This is somewhat similar to group convolutions, which partitions a large convolution into smaller convolutions. The increase in communication channels in a neural network setting allows the different vertices in our sequence graphs to store more information and update the queries and keys accordingly. For instance, there are underlying patterns for a single letter 'a', which is a vowel, a common starting letter, and can be followed by many letters and have many positions. The patterns that occur for 'a' in practice would be very cumbersome to explicitly state, so we learn and update the weights of the W_Q and W_K transformations in order to mimic these patterns. This is the power of neural networks.

We need to mention the possibility of overfitting since these additional layer implementations dramatically increase the parameter counts. If training and you see the validation loss is any less than the evaluation loss, you have overfitting. One way to reduce this is by adding dropout after certain large computations such as attention blocks in order to create subnets of communication channels. This prevents overfitting since communication channels will not have complete access to all communication channels.

17.2 Feed-Forward Component

The (single or multi-) attention block can be passed to a simple feed-forward neural network before it is projected to the logits space, i.e.,

$$ReLU\left(xW_1 + b_1\right)W_2 + b_2$$

in which x is the attention block. The reason we use two linear transformations is so that we can increase the dimensional inner layers by some positive scalar and then scale back down to the embedding size, i.e., if the size of the embedding is m and we scale by 4, then $W_1 \in (m, 4m)$ and $W_2 \in (4m, m)$.

17.3 Residual Connections

Recall 1 proved addition operations do not affect the gradients.

17.4 LayerNormalization

Layer normalization is a normalization operation that makes its feature inputs to unit Gaussian. It occurs across the rows instead of columns like batch normalization, and does not require running mean or running variance. As of this writing in 2025, the layer normalization is applied before the attention transformations and before the feed-forward component. This is called **pre-form normalization**. The original implementation of transformers in "Attention is all you need" added it after. The layerNorm's size equals the embedding size.

Part VI
References:

- https://leimao.github.io/blog/Maximum-Likelihood-Estimation-Ngram/
- https://math.stackexchange.com/questions/1763428/integral-tanh-and-normal
- https://math.stackexchange.com/questions/1669735/expectation-of-a-standard-normal-random-variable