LLM

Name

05/08/2025 11:13

1 Building Makemore

1.1 Bigrams model: predict next char

From bigrams, count the number of appearances of a pair to produce a probability distribution of the likelihood that a pair appears given the first character. We can simply draw a sample from it. One problem is that some pairs have a count of 0. When we calculate the log-likelihood, this can be an issue. We can add a "fake" count of 1.

1.2 Neural Network approach

We can create a training set by passing indices to tensors, encoding them using one-hot encoding. Using weights, we can project the previous character to the count table of the next character.

$$x_{\text{encd}} \times W = \log(\text{no of counts})$$

$$P = \frac{C}{C.\text{Sum}()} = \text{probability of the next character}$$

We use a training sample to optimize the weights. The final result will be exactly the counting table.

1.3 MLP

The bigram approach can only work with a single previous character and is not extendable if we want to use more context. We can associate each word in the vocabulary with a feature vector. We can train a neural network to embed words with similar meanings into a similar space. The proposed approach can be summarized as follows:

1. Associate with each word in the vocabulary a distributed word feature vector (a real-valued vector in $\mathbb{R}^m$). The number of features (e.g. m =30, 60 or 100 in the experiments) is much smaller than V - the size of the vocabulary (e.g. 17,000) or size of the alphabet (e.g. 26).

2. Express the joint probability function of word sequences in terms of the feature vectors of the words in the sequence.

3. Learn simultaneously the word feature vectors and the parameters of that probability function.

"Similar" words are expected to have a similar feature vector, and because the probability function is a smooth function of these feature values, a small change in the features will induce a small change in the probability. Therefore, the presence of only one of the above sentences in the training data will increase the probability, not only of that sentence, but also of its combinatorial number of "neighbors" in sentence space (as represented by sequences of feature vectors).

1.3.1 A Neural Model

The training set is a sequence $w_1, \ldots, w_T$ of words $w_t \in V$, where the vocabulary V is a large but finite set. The objective is to learn a good model

$$f(w_t, \ldots, w_{t-n+1}) = \hat{P}(w_t \mid w_1^{t-1}).$$

We decompose the function $f(w_t, \ldots, w_{t-n+1}) = \hat{P}(w_t \mid w_1^{t-1})$ into two parts:

1. A mapping C from any element i of V to a real vector $C(i) \in \mathbb{R}^m$. It represents the distributed feature vectors associated with each word in the vocabulary. In practice, C is represented by a $|V| \times m$ matrix of free parameters.

2. The probability function over words, expressed with C: a function g maps an input sequence of feature vectors for words in context, $(C(w_{t-n+1}), \ldots, C(w_{t-1}))$, to a conditional probability distribution over words in V for the next word w_t. The output of g is a vector whose i-th element estimates the probability $\hat{P}(w_t = i \mid w_1^{t-1})$.

$$f(i, w_{t-1}, \ldots, w_{t-n+1}) = g(i, C(w_{t-1}), \ldots, C(w_{t-n+1}))$$

The function g may be implemented by a feed-forward or recurrent neural network, or another parametrized function, with parameters ω. The overall parameter set is $\theta = (C, \omega)$. Training is achieved by looking for θ that maximizes the penalized log-likelihood over the training corpus:

$$\mathcal{L} = \frac{1}{T} \sum_t \log f(w_t, w_{t-1}, \ldots, w_{t-n+1}; \theta) + R(\theta),$$

where $R(\theta)$ is a regularization term. For example, in our experiments, R is a weight decay penalty applied only to the weights of the neural network and to the C matrix, not to the biases. the number of free parameters only scales linearly with V, the number of words in the vocabulary. It also only scales linearly with the order n.

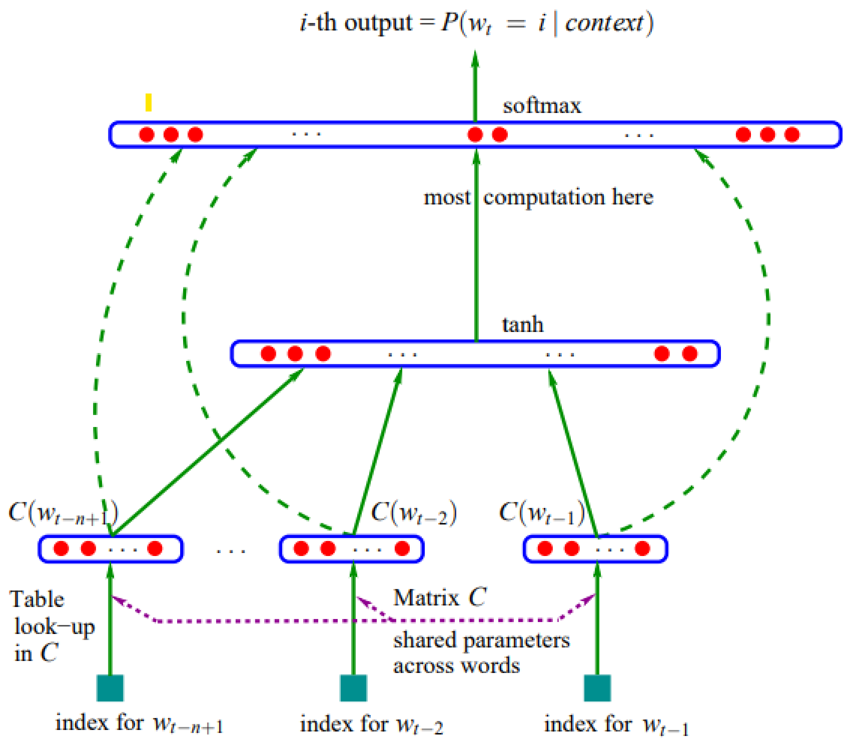

Figure 1: **Neural architecture:** $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$, where g is the neural network and $C(i)$ is the i-th word feature vector.

1.3.2 Neural Network Common Issues

Initial Loss is usually too high if we just start with set of random numbers. The initial loss should be based on an intial guess of the solution. For example, when predicting the next character, the initial loss should be based on the uniform probability of guressing the correct character (1/27). In theory, the weights W_2 should be initialized to zero, since $\log(1) = 0$, corresponding to an initial count of one for each character. However, if the final weight W is set to zero, the output of the tanh layer effectively becomes the final output of the network. Consequently, computing the gradient becomes problematic, as the gradient would vanish. Recall that the backward derivative of $\tanh(t)$ is $(1-t^2) \cdot \frac{\partial L}{\partial out}$, and the tanh activation function squeezes many values toward -1 and 1, leading to zero gradients. Similar issues arise for the sigmoid and ReLU activation functions.

To avoid the **saturated tanh problem**, we can scale the weights before passing into activation function. For example, We now employ Kaiming initialization for the weights W_1: $\frac{5/3}{\sqrt{n_{in}}}$. The gain factor of 5/3 for the tanh activation arises from the expected value of $\tanh^2(x)$, where x follows a standard Gaussian distribution: $\int_{-\infty}^{\infty} (\tanh x)^2 \frac{\exp(-x^2/2)}{\sqrt{2\pi}}\, dx \approx 0.39$. The square root of this value quantifies the extent to which tanh compresses the variance of the input variable: $0.39^{0.5} \approx 0.63 \approx 3/5$. Consequently, to preserve an output variance of 1, we scale by the gain 5/3.

However, with deep neural network, scaling all the weights become troublesome, so we can use **batch normalisation** istead. Due to the mean subtraction in batch normalization, which centers the activations, the bias term in the preceding linear layer becomes redundant and can be omitted. For a mini-batch $\mathcal{B} = \{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\}$ of size m, compute the batch mean and variance along the batch dimension:

$$\boldsymbol{\mu}_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^{m} \mathbf{z}^{(i)},$$

$$\boldsymbol{\sigma}_{\mathcal{B}}^2 = \frac{1}{m} \sum_{i=1}^{m} (\mathbf{z}^{(i)} - \boldsymbol{\mu}_{\mathcal{B}})^2.$$

The normalized pre-activation is then obtained by:

$$\hat{\mathbf{z}}^{(i)} = \frac{\mathbf{z}^{(i)} - \boldsymbol{\mu}_{\mathcal{B}}}{\sqrt{\boldsymbol{\sigma}_{\mathcal{B}}^2 + \epsilon}},$$

$$\mathbf{y}^{(i)} = \boldsymbol{\gamma} \odot \hat{\mathbf{z}}^{(i)} + \boldsymbol{\beta},$$

where $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learnable parameters (scale and shift, respectively), $\odot$ denotes element-wise multiplication, and $\epsilon > 0$ is a small constant (e.g., 10^{-5}) added for numerical stability to prevent division by zero when the batch variance is very small. This affine transformation allows the layer to recover the identity function if needed, preserving representational capacity.

During training, batch-specific statistics are used, but at inference time, we rely on population statistics estimated from the entire training set to enable normalization for arbitrary input sizes (e.g., single examples). These are maintained as running estimates via an exponential moving average (EMA), updated without gradient computation:

$$\begin{aligned}&\text{with } \texttt{torch.no\_grad()} :\\ &\quad \boldsymbol{\mu}_{\text{running}} = (1-\tau)\cdot\boldsymbol{\mu}_{\text{running}} + \tau\cdot\boldsymbol{\mu}_{\mathcal{B}},\\ &\quad \boldsymbol{\sigma}^2_{\text{running}} = (1-\tau)\cdot\boldsymbol{\sigma}^2_{\text{running}} + \tau\cdot\boldsymbol{\sigma}^2_{\mathcal{B}}.\end{aligned}$$

Here, τ is the momentum parameter set at 0.1 by default in Pytorch.

1.3.3 Usual Plots to investigate Neural Network Performance

To investigate common issues with Neural Network, we can use the following plots:

- Loss function graph, too much volatility may imply the batch is too small
- Activation output distributions across layers - to check if the activation function produce extreme outputs (for example it's not good if tanh produces a lot of values close to 1,-1)
- Gradient distribution across different layers, a good graph would be the gradient behaved similarly in different layers, not shrinking or exploding.
- Weights Gradiennt Distribtion.
- Gradient to Data ratio during update - a good value would be 1e-3, otherwise the update might be too slow or too fast.

1.3.4 Quick Think on how this can be applied to financial series

- **Words are discrete:** Natural language processing benefits from a known, finite vocabulary, allowing each word to be represented as a discrete token.
- **Finance is continuous:** Financial data has no predefined vocabulary and can take on a continuous range of values due to its high variability and complexity.
- **Modeling challenge:** The absence of a discrete vocabulary in finance raises the problem of how to effectively represent continuous data in a way that supports learning and generalization.

To address

Clustering techniques: Methods like *k-means*, *Gaussian Mixture Models*, or *Hidden Markov Models (HMMs)* can be used to group financial observations into a finite number of states or regimes.

Vector quantization: Transforming continuous state spaces into discrete representations using quantization techniques can help bridge the gap between continuous inputs and discrete models.

Learned embeddings: Use neural networks (e.g., autoencoders or contrastive learning) to learn compressed, structured representations of financial states that can be mapped to discrete clusters or classes.

1.4 Backpropagation Example

1.4.1 Forward Pass

Variable Definitions

- n: Number of samples in a batch
- k: Context size (e.g., number of characters in a sequence)
- m: Number of features / embedding dimension
- V: Vocabulary size
- n_1: Number of neurons in the hidden layer
- $C \in \mathbb{R}^{V \times m}$: Embedding matrix
- $X_b \in \mathbb{R}^{n \times k}$: Input batch of character indices
- $Y_b \in \mathbb{R}^{n}$: Target character indices

Embedding Layer The embedding layer maps each input index to a dense vector:

$$\begin{aligned} \text{emb} &= C[X_b], && \text{Shape: } \mathbb{R}^{n \times k \times m} \\ \text{emb}_{\text{cat}} &= \text{emb.view}(n, k \times m), && \text{Shape: } \mathbb{R}^{n \times (k \cdot m)} \end{aligned}$$

Linear Layer 1 The first hidden layer (pre-batch-normalization):

$$h_{\text{prebn}} = \text{emb}_{\text{cat}} W_1 + b_1, \qquad W_1 \in \mathbb{R}^{(k \cdot m) \times n_1}, \quad b_1 \in \mathbb{R}^{1 \times n_1}, \quad \text{Shape: } \mathbb{R}^{n \times n_1}$$

Batch Normalization Layer

1. **Mean:**
$$\text{bmean} = \frac{1}{n} \sum_{i=1}^{n} (h_{\text{prebn}})_i, \quad \text{Shape: } \mathbb{R}^{1 \times n_1}$$

2. **Center:**
$$\text{bndiff} = h_{\text{prebn}} - \text{bmean}, \quad \text{Shape: } \mathbb{R}^{n \times n_1}$$

3. **Variance:**

$$\begin{aligned} \text{bndiff}^2 &= \text{bndiff} \odot \text{bndiff}, && \text{Shape: } \mathbb{R}^{n\times n_1} \\ \text{bnvar} &= \frac{1}{n-1}\sum_{i=1}^{n}(\text{bndiff}^2)_i, && \text{Shape: } \mathbb{R}^{1\times n_1} \end{aligned}$$

4. **Normalize:**

$$\begin{aligned} \text{bnvar\_inv} &= \frac{1}{\sqrt{\text{bnvar}+\epsilon}}, && \text{Shape: } \mathbb{R}^{1\times n_1} \\ \text{bnraw} &= \text{bndiff} \odot \text{bnvar\_inv}, && \text{Shape: } \mathbb{R}^{n\times n_1} \end{aligned}$$

5. **Scale and Shift:**

$$h_{\text{preact}} = \text{bngain} \odot \text{bnraw} + \text{bnbias}, \quad \text{Shape: } \mathbb{R}^{n\times n_1}$$

Tanh Activation

$$h = \tanh(h_{\text{preact}}), \quad \text{Shape: } \mathbb{R}^{n\times n_1}$$

1.4.2 Output Layer & Loss

1. **Final Linear Layer:**

$$\text{logits} = hW_2 + b_2, \qquad W_2 \in \mathbb{R}^{n_1\times V}, \quad b_2 \in \mathbb{R}^{1\times V}, \quad \text{Shape: } \mathbb{R}^{n\times V}$$

2. **Softmax (Stable):**

$$\begin{aligned} \text{logit\_maxes} &= \max_j(\text{logits}_{ij}), && \text{Shape: } \mathbb{R}^{n\times 1} \\ \text{norm\_logits} &= \text{logits} - \text{logit\_maxes}, && \text{Shape: } \mathbb{R}^{n\times V} \\ \text{counts} &= \exp(\text{norm\_logits}), && \text{Shape: } \mathbb{R}^{n\times V} \\ \text{count\_sum} &= \sum_{j=1}^{V}\text{counts}_{ij}, && \text{Shape: } \mathbb{R}^{n\times 1} \\ \text{probs} &= \frac{\text{counts}}{\text{count\_sum}}, && \text{Shape: } \mathbb{R}^{n\times V} \end{aligned}$$

3. **Cross-Entropy Loss:**

$$\begin{aligned} \text{logprobs} &= \log(\text{probs}), && \text{Shape: } \mathbb{R}^{n\times V} \\ L &= -\frac{1}{n}\sum_{i=1}^{n}\text{logprobs}[i, Y_{b_i}], && \text{Scalar} \end{aligned}$$

1.4.3 Backward Pass

Gradients w.r.t. Logits

1. ***d*logprobs:**

$$d\text{logprobs}_{ij} = \begin{cases} -\frac{1}{n}, & j = Y_{b_i} \\ 0, & \text{otherwise} \end{cases} \quad \text{Shape: } \mathbb{R}^{n \times V}$$

2. ***d*probs:**

$$d\text{probs} = d\text{logprobs} \odot \frac{1}{\text{probs}}, \quad \text{Shape: } \mathbb{R}^{n \times V}$$

3. **Counts and sums:**

$$\begin{aligned} d\text{count\_sum\_inv} &= \sum_j (d\text{probs} \odot \text{counts})_j, && \text{Shape: } \mathbb{R}^{n \times 1} \\ d\text{count\_sum} &= d\text{count\_sum\_inv} \odot \frac{-1}{\text{count\_sum}^2}, && \mathbb{R}^{n \times 1} \\ d\text{counts} &= d\text{probs} \odot \frac{1}{\text{count\_sum}} + d\text{count\_sum} \odot I, && \mathbb{R}^{n \times V}, I : \mathbb{R}^{n \times V} \end{aligned}$$

4. ***d*logits:**

$$\begin{aligned} d\text{norm\_logits} &= d\text{counts} \odot \exp(\text{norm\_logits}), && \mathbb{R}^{n \times V} \\ d\text{logit\_maxes} &= \sum_j d\text{norm\_logits}_{ij}, && \mathbb{R}^{n \times 1} \\ d\text{logits} &= d\text{norm\_logits} + d\text{logit\_maxes} \cdot \mathbb{I}(j = \text{argmax}_k \text{logits}_{ik}), && \mathbb{R}^{n \times V} \end{aligned}$$

Gradients w.r.t. Hidden Layer

$$\begin{aligned} dW_2 &= h^T d\text{logits}, && \mathbb{R}^{n_1 \times V} \\ db_2 &= \sum_{i=1}^{n} (d\text{logits})_i, && \mathbb{R}^{1 \times V} \\ dh &= d\text{logits} W_2^T, && \mathbb{R}^{n \times n_1} \\ dh_{\text{preact}} &= dh \odot (1 - h^2), && \mathbb{R}^{n \times n_1} \end{aligned}$$

Gradients w.r.t. Batch Norm

$$
\begin{aligned}
d\text{bngain} &= \sum_{i=1}^{n} (dh_{\text{preact}} \odot \text{bnraw})_i, && \mathbb{R}^{1\times n_1} \\
d\text{bnbias} &= \sum_{i=1}^{n} (dh_{\text{preact}})_i, && \mathbb{R}^{1\times n_1} \\
d\text{bnraw} &= dh_{\text{preact}} \odot \text{bngain}, && \mathbb{R}^{n\times n_1} \\
d\text{bnvar\_inv} &= \sum_{i=1}^{n} (d\text{bnraw} \odot \text{bndiff})_i, && \mathbb{R}^{1\times n_1} \\
d\text{bnvar} &= d\text{bnvar\_inv} \odot \frac{-1}{2}(\text{bnvar} + \epsilon)^{-3/2}, && \mathbb{R}^{1\times n_1} \\
d\text{bndiff} &= d\text{bnraw} \odot \text{bnvar\_inv} + \frac{2}{n-1}\text{bndiff} \odot d\text{bnvar}, && \mathbb{R}^{n\times n_1} \\
dh_{\text{prebn}} &= d\text{bndiff} - \frac{1}{n}\sum_{i=1}^{n} (d\text{bndiff})_i, && \mathbb{R}^{n\times n_1}
\end{aligned}
$$

Gradients w.r.t. Embedding Layer

$$
\begin{aligned}
dW_1 &= \text{emb}_{\text{cat}}^{T} dh_{\text{prebn}}, && \mathbb{R}^{(k\cdot m)\times n_1} \\
db_1 &= \sum_{i=1}^{n} (dh_{\text{prebn}})_i, && \mathbb{R}^{1\times n_1} \\
d\text{emb}_{\text{cat}} &= dh_{\text{prebn}} W_1^{T}, && \mathbb{R}^{n\times(k\cdot m)} \\
d\text{emb} &= d\text{emb}_{\text{cat}}.\text{view}(n,k,m), && \mathbb{R}^{n\times k\times m}
\end{aligned}
$$

Finally, update C:

$$
dC[j] \mathrel{+}= \sum_{i,l\,:\,X_{b_{il}}=j} d\text{emb}_{il}
$$

1.4.4 Softmax-Loss Gradient Direct calculation

Given Loss $L = -\frac{1}{N}\sum_{i=1}^{N}\log(P_{iy_i})$ where $P_{ij} = \frac{\exp(z_{ij})}{\sum_k \exp(z_{ik})}$. We want $\frac{\partial L}{\partial z_{ij}}$.

Case 1: $j = y_i$ **(derivative w.r.t. the correct logit)**

$$\begin{aligned}
\frac{\partial}{\partial z_{iy_i}}(-\log P_{iy_i}) &= -\frac{1}{P_{iy_i}}\frac{\partial P_{iy_i}}{\partial z_{iy_i}} \\
&= -\frac{1}{P_{iy_i}}\frac{\exp(z_{iy_i})(\sum_k \exp(z_{ik})) - \exp(z_{iy_i})\exp(z_{iy_i})}{(\sum_k \exp(z_{ik}))^2} \\
&= -\frac{1}{P_{iy_i}}\left(\frac{\exp(z_{iy_i})}{\sum_k \exp(z_{ik})} - \left(\frac{\exp(z_{iy_i})}{\sum_k \exp(z_{ik})}\right)^2\right) \\
&= -\frac{1}{P_{iy_i}}(P_{iy_i} - P_{iy_i}^2) = -(1 - P_{iy_i}) \\
&= P_{iy_i} - 1
\end{aligned}$$

Case 2: $j \neq y_i$ **(derivative w.r.t. an incorrect logit)**

$$\begin{aligned}
\frac{\partial}{\partial z_{ij}}(-\log P_{iy_i}) &= -\frac{1}{P_{iy_i}}\frac{\partial P_{iy_i}}{\partial z_{ij}} \\
&= -\frac{1}{P_{iy_i}}\frac{-\exp(z_{iy_i})\exp(z_{ij})}{(\sum_k \exp(z_{ik}))^2} \\
&= \frac{1}{P_{iy_i}}\left(\frac{\exp(z_{iy_i})}{\sum_k \exp(z_{ik})} \cdot \frac{\exp(z_{ij})}{\sum_k \exp(z_{ik})}\right) \\
&= \frac{1}{P_{iy_i}}(P_{iy_i} \cdot P_{ij}) \\
&= P_{ij}
\end{aligned}$$

Combining these cases (and averaging over the batch of size n), the gradient matrix $d$logits is $\frac{1}{n}(P - Y)$, where Y is the one-hot matrix of true labels.

1.4.5 Batch normalisation Gradient Direct Calculation

1. **Mean (μ):**

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

2. **Variance (σ^2):**

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2$$

3. **Normalization ($\hat{x}_i$):** The input is normalized using the batch statistics.

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

4. **Scale and Shift (y_i):** The normalized activation is scaled by γ and shifted by β.
$$y_i = \gamma \hat{x}_i + \beta$$

The goal is to compute the gradient of the loss L with respect to the input of the layer, $\frac{\partial L}{\partial x_i}$. This is derived by applying the chain rule, which requires finding how the loss changes with respect to the intermediate variables $\hat{x}_i$, μ, and σ^2.

First, the gradient with respect to the normalized activation $\hat{x}_i$ is found, where $\frac{\partial L}{\partial y_i}$ is the incoming gradient from the subsequent layer.

$$\frac{\partial L}{\partial \hat{x}_i} = \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial \hat{x}_i} = \frac{\partial L}{\partial y_i}\gamma$$

Next, the gradients for the batch statistics are found by summing the contributions from all activations in the batch.

$$\frac{\partial L}{\partial \sigma^2} = \sum_{i=1}^{n} \frac{\partial L}{\partial \hat{x}_i}\frac{\partial \hat{x}_i}{\partial \sigma^2}$$
$$\frac{\partial L}{\partial \mu} = \sum_{i=1}^{n} \left(\frac{\partial L}{\partial \hat{x}_i}\frac{\partial \hat{x}_i}{\partial \mu}\right) + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial \mu}$$

Finally, the total derivative for the input x_i is the sum of the gradients from the three computational paths it affects: its direct path, its contribution to the mean, and its contribution to the variance.

$$\begin{aligned}\frac{\partial L}{\partial x_i} = & \frac{\partial L}{\partial \hat{x}_i}\frac{\partial \hat{x}_i}{\partial x_i} && \text{(Path 1: Direct influence on } \hat{x}_i\text{)} \\ & + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial x_i} && \text{(Path 2: Influence via the mean)} \\ & + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial x_i} && \text{(Path 3: Influence via the variance)}\end{aligned}$$

1.5 Simple Wavenet Implementation

Instead of passing all the context into embedding at once, we can split the input data into small sequence length first. The idea is illustrated in 1.5.

The input data will now have 3 dimensions (N, L, C) where N is the batch size, L is the sequence length and C is the number of features or channel. When doing batch normalisation, ideally we would compute the mean across the dimension (N,L). However, pyTorch batchnorm assumes the input is (N, C, L) format so the mean is reduced differently.

2 Tokenization

Modern Large Language Models (LLMs) do not operate directly on raw text. Instead, text is converted into discrete tokens drawn from a finite vocabulary.

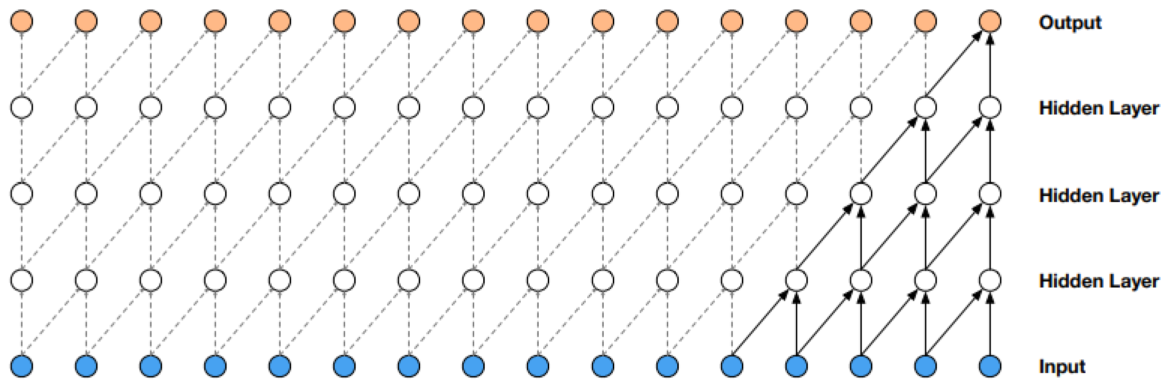

Figure 2: Visualization of a stack of causal convolutional layers

.

Understanding this process is crucial for both interpreting model behavior and designing efficient tokenization schemes for new domains (e.g., financial data).

2.1 From Bytes to Tokens

GPT-style tokenization begins with a base vocabulary of 256 raw byte tokens (0–255), ensuring that *any* text can be represented. From this starting point, **Byte Pair Encoding (BPE)** merges the most frequent adjacent tokens into new tokens, gradually learning common words and subwords.

However, naive BPE can be inefficient:

- It may create many variants of the same word (e.g., `dog.`, `dog!`, `dog?`), wasting vocabulary slots.
- Spaces, punctuation, and multi-byte characters complicate merging rules.

To address this, GPT introduces a **regex-based pre-tokenizer** before applying BPE merges.

2.2 Regex Pre-Tokenizer Design

The pre-tokenizer applies two key rules:

- **Spaces are glued to the following word** (`?\p{L}+`). This avoids doubling sequence length by predicting "space" + word separately. For example, `"dog"` and `" dog"` are treated as distinct tokens.
- **Punctuation is always split** (`?[^\s\p{L}\p{N}]+`). This prevents wasteful tokens like `dog.`, `dog!`, `dog?`, instead reusing `"dog"` + punctuation.

The design trade-off is:

$$\text{spaces are extremely frequent} \;\rightarrow\; \text{glue them}, \quad \text{punctuation is diverse and sparse} \;\rightarrow\; \text{split it.}$$

This balances vocabulary efficiency with manageable sequence length.

2.3 Concrete Examples

Experiments in the notebook illustrate how different tokenizers (GPT-2 vs. GPT-4's `cl100k_base`) handle spaces and punctuation.

Input Text	**GPT-2 Tokens**	**GPT-4 Tokens**
`"dog."`	`["dog."]`	`["dog", "."]`
`"dog!"`	`["dog!"]`	`["dog", "!"]`
`" hello"`	4 separate space tokens + `"hello"`	merged into one token with spaces
`" world"`	`[" world"]`	`[" world"]`

Key observations:

- GPT-2 leaves each space as a separate token, while GPT-4 merges consecutive spaces into a single token.
- Both models glue spaces to words, making `"dog"` and `" dog"` different tokens.
- Punctuation is separated in GPT-4, improving vocabulary efficiency.

2.4 Beyond Text: Financial Applications

Tokenization concepts extend beyond natural language. In finance, raw numbers (interest rates, prices) can be quantized or bounded:

- Rates can be restricted to plausible ranges (e.g., 0–5%).
- Values may be rounded to two decimal places or basis points.

Even with such constraints, the vocabulary size may remain large, so compression techniques similar to BPE are often required.

In summary, tokenization is a balance between *expressiveness* (capturing frequent patterns in fewer tokens) and *efficiency* (avoiding unnecessary vocabulary growth). The regex pre-tokenizer in GPT demonstrates how careful design of rules for spaces and punctuation leads to practical improvements in model training and performance.

3 Attention Mechanism and Transformer Architecture

We present the Transformer model for next-token prediction, including the core architecture and the mathematical reasoning behind several stabilisation techniques used in implementation.

3.1 Problem Setup

We train a character-level language model: given a context of T characters, the model predicts the next token.

- Vocabulary size V: number of unique tokens.
- Block size T: context length.
- Batch size B: number of sequences per batch.
- Embedding dimension C: dimension of token embeddings.

The input has shape $B \times T$, mapped to embeddings of shape $B \times T \times C$. The model outputs logits of shape $B \times T \times V$.

The training loss is average cross-entropy:

$$\mathcal{L} = -\frac{1}{BT} \sum_{b=1}^{B} \sum_{t=1}^{T} \log \hat{y}_{b,t}[y_{b,t}],$$

where $\hat{y}_{b,t} = \mathrm{softmax}(o_{b,t})$ are predicted probabilities.

3.2 Embeddings and Weight Tying

Each token index x_t is mapped to a vector:

$$h_t^{(0)} = E[x_t] + P_t,$$

with token embeddings $E \in \mathbb{R}^{V \times C}$ and positional embeddings $P_t \in \mathbb{R}^C$.

We apply *weight tying*: the same E is reused for output:

$$\mathrm{logits}_t = h_t^{(L)} E^\top,$$

reducing parameters and aligning input/output spaces.

3.3 Self-Attention

For hidden states $h \in \mathbb{R}^{T \times C}$, queries, keys, and values are

$$Q = hW_Q, \quad K = hW_K, \quad V = hW_V,$$

with $W_Q, W_K, W_V \in \mathbb{R}^{C \times H}$ for head size H.

Attention scores:

$$A = QK^\top.$$

Scaling. If Q, K entries have variance 1, then $\mathrm{Var}[QK^\top] \approx H$. Large H yields extreme logits, saturating softmax. We stabilise by scaling:

$$\tilde{A} = \frac{QK^\top}{\sqrt{H}}.$$

Causal Masking. To enforce autoregression:

$$\tilde{A}_{ij} = \begin{cases} \frac{Q_i K_j^\top}{\sqrt{H}}, & j \le i, \\ -\infty, & j > i, \end{cases}$$

so token i cannot attend to the future. In practice, we use PyTorch's fused scaled dot-product attention with `is_causal=True`, avoiding explicit mask buffers.

Weighted Sum. Attention weights:

$$W = \mathrm{softmax}(\tilde{A}),$$

and outputs:

$$O = WV.$$

3.4 Multi-Head Attention

With M heads:

$$\mathrm{MHA}(h) = \big[O^{(1)} \,\|\, \cdots \,\|\, O^{(M)}\big] W_O,$$

where $W_O \in \mathbb{R}^{MH \times C}$. Multiple heads allow capturing different dependency structures.

3.5 Feedforward Network

Each block also includes a positionwise MLP:

$$\mathrm{FFN}(h) = \sigma(hW_1 + b_1)W_2 + b_2,$$

with hidden dimension $4C$ and nonlinearity σ (GELU or ReLU).

3.6 Residual Connections and Normalization

Residuals:

$$h^{(\ell+1)} = h^{(\ell)} + \frac{1}{\sqrt{2n}} f(h^{(\ell)}),$$

where f is MHA or FFN, and n is the number of residual blocks. The scaling factor prevents residual variance from growing with depth.

LayerNorm is applied before each sub-layer to stabilise activations. Unlike BatchNorm, it normalises across features within each token, making it suitable for sequences.

3.7 Training Details and Stabilisation Tricks

Initialization. Linear layers are initialised as

$$W_{ij} \sim \mathcal{N}(0, 0.02^2).$$

Final projections are additionally scaled down by depth to avoid residual explosions.

Optimizer (AdamW). Update rule:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{m_t}{\sqrt{v_t} + \epsilon} - \eta \cdot \lambda \theta_t,$$

with moving averages m_t, v_t and decay coefficient λ. Biases and LayerNorm weights are excluded from decay for stability.

Learning Rate Schedule. Warmup + cosine decay:

$$\eta_t = \begin{cases} \eta_{\max} \cdot \frac{t}{s_w}, & t \le s_w, \\ \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos \frac{\pi(t - s_w)}{T - s_w}), & t > s_w, \end{cases}$$

with warmup steps s_w and total steps T.

Mixed Precision. Training is run in fp16 for speed, with critical ops (softmax, LayerNorm) in fp32 to avoid numerical issues.

Gradient Clipping. To prevent explosions:

$$g \mapsto \frac{g}{\max(1, \|g\|_2 / \tau)}, \quad \tau = 1.0.$$

Validation Monitoring. Data is split into train/val. Validation loss:

$$\mathcal{L}_{\text{val}} = \frac{1}{B_{\text{val}} T} \sum_{b=1}^{B_{\text{val}}} \sum_{t=1}^{T} - \log \hat{y}_{b,t}[y_{b,t}],$$

is tracked to detect overfitting.

3.8 Summary

The full model consists of:

1. Token + positional embeddings (with weight tying).
2. Stacked Transformer blocks (MHA + FFN + residuals + LayerNorm).
3. Output projection to vocabulary size.

Stability is ensured through scaled attention, careful init, residual scaling, AdamW with selective decay, warmup+cosine scheduling, mixed precision, gradient clipping, and validation monitoring.

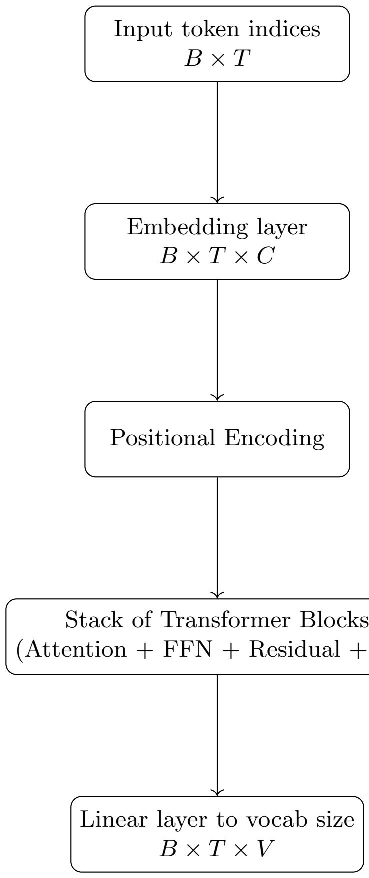

Figure 3: Flowchart of the character-level Transformer network.

3.9 Practical Note and Link to GPT

- In practice, the tokenization is not done on character level but on sub word. Hence, the vocabulary size is typically 10-100k tokens instead of just 65 in our problem.
- The context size for GPT-3 is 2048 token.
- After pretrainning, the base model only wants to complete documents not answering questions.
- The model is then fine-tunned for specific model using Reward Modeling and Reinforcement Learning.
- For financial application, probably a cross attention is required (encoder is for controlled variables and decoder is for the time series of variable itself)

4 Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting

In application to time series, Transformer allows the model to access any part of the histori regardless of distance, making it potentially more suitable for grasping the recurring patterns with long-term dependencies. However, space complexilty of Transfor grows quadratically with the input (sequence length) T, which causes memory Bottleneck on directly modeling long time series with fine granualirty.

Problem definition We consider a collection of N related univariate time series

$$\{z_{i,1:t_0}\}_{i=1}^{N}, \quad z_{i,1:t_0} = [z_{i,1}, z_{i,2}, \dots, z_{i,t_0}], \quad z_{i,t} \in \mathbb{R},$$

where $z_{i,t}$ denotes the value of series i at time t. The goal is to predict the next τ steps for all series, i.e.

$$\{z_{i,t_0+1:t_0+\tau}\}_{i=1}^{N}.$$

Each series is associated with known time-based covariates

$$\{x_{i,1:t_0+\tau}\}_{i=1}^{N}, \quad x_{i,t} \in \mathbb{R}^d,$$

such as day-of-week or hour-of-day. We aim to model the conditional distribution

$$p(z_{i,t_0+1:t_0+\tau} \mid z_{i,1:t_0}, x_{i,1:t_0+\tau}; \Phi) = \prod_{t=t_0+1}^{t_0+\tau} p(z_{i,t} \mid z_{i,1:t-1}, x_{i,1:t}; \Phi),$$

where Φ denotes shared learnable parameters. This factorization follows from the chain rule of probability and does not assume future time steps are independent; each term conditions on the entire past trajectory.

Thus, the problem reduces to learning a one-step-ahead predictive model

$$p(z_t \mid z_{1:t-1}, x_{1:t}; \Phi).$$

To incorporate both past observations and covariates, we define the augmented vector

$$y_t = [\, z_{t-1} \circ x_t \,] \in \mathbb{R}^{d+1}, \quad Y_t = [\, y_1, \dots, y_t \,]^\top \in \mathbb{R}^{t \times (d+1)},$$

where $[\cdot \circ \cdot]$ denotes concatenation. The task is then to explore models of the form

$$z_t \sim f(Y_t),$$

that predict the distribution of z_t given the augmented history Y_t.

Transformer We instanitate f with Transformer. The input to the self-attention layer is the augmented history matrix (here, we drop the t subscript in Y for ease of notation)

$$Y \in \mathbb{R}^{t \times (d+1)},$$

where t denotes the sequence length and $d + 1$ is the feature dimension at each time step (one observation plus d covariates). In practice, one often processes multiple time series in parallel, in which case Y generalizes to $Y \in \mathbb{R}^{B \times t \times (d+1)}$ with batch size B. This setup is analogous to text Transformers, where $d + 1$ plays the role of the embedding dimension d_{model}.

For each attention head $h = 1, \ldots, H$, we compute queries, keys, and values via learned linear projections:

$$Q_h = YW_h^Q, \quad K_h = YW_h^K, \quad V_h = YW_h^V,$$

where

$$W_h^Q, W_h^K \in \mathbb{R}^{(d+1) \times d_k}, \quad W_h^V \in \mathbb{R}^{(d+1) \times d_v}.$$

Here d_k is the query/key dimension per head and d_v is the value dimension per head. Typically $d_k = d_v = d_{\text{model}}/H$, ensuring that the concatenated multi-head output has dimension d_{model}, but in principle d_k and d_v may differ.

The output of head h is given by

$$O_h = \text{Attention}(Q_h, K_h, V_h) = \text{softmax}\left(\frac{Q_h K_h^\top}{\sqrt{d_k}} + M\right) V_h,$$

where

- $Q_h K_h^\top \in \mathbb{R}^{t \times t}$ contains all pairwise dot products,
- $M \in \mathbb{R}^{t \times t}$ is the causal mask (upper-triangular entries set to $-\infty$),
- softmax is applied row-wise,
- thus $O_h \in \mathbb{R}^{t \times d_v}$.

The outputs from all heads are concatenated and linearly projected:

$$O = \text{Concat}(O_1, \ldots, O_H)\, W^O,$$

with $W^O \in \mathbb{R}^{Hd_v \times (d+1)}$ learnable and $O \in \mathbb{R}^{t \times (d+1)}$.

Each position (time step) is then processed independently by a two-layer feedforward network:

$$\text{FFN}(z) = \sigma(zW_1 + b_1)W_2 + b_2,$$

where

- $z \in \mathbb{R}^{1 \times (d+1)}$ is a single row of O,
- $W_1 \in \mathbb{R}^{(d+1) \times d_{\text{ff}}}, \quad b_1 \in \mathbb{R}^{d_{\text{ff}}}$,

- $W_2 \in \mathbb{R}^{d_{\text{ff}} \times (d+1)}, \quad b_2 \in \mathbb{R}^{d+1}$,
- $\sigma(\cdot)$ is the ReLU activation.

Applying this transformation to every row of O yields

$$\text{FFN}(O) \in \mathbb{R}^{t \times (d+1)}.$$

4.1 Enhancing the Locality of Transformer

Patterns in time series may evolve with time significantly due to various events, so whether an observed point is an anomaly, change point or part of the patterns is highly dependent on its surrounding context. However, in the self-attention layers, the similiarities between queries and keys are computed based on their point-wise values without fully leveraging local context.

Convolutional Transformer We instantiate f with a Convolutional Transformer. The input to the self-attention layer is the augmented history matrix

$$Y \in \mathbb{R}^{t \times (d+1)},$$

where t denotes the sequence length and $d+1$ the feature dimension at each time step. For multiple parallel time series, we have $Y \in \mathbb{R}^{B \times t \times (d+1)}$ with batch size B.

For each attention head $h = 1, \dots, H$, the queries and keys are computed via causal convolution over a local window of size k:

$$Q_h = \text{Conv1D}_h^Q(Y), \quad K_h = \text{Conv1D}_h^K(Y), \quad V_h = YW_h^V,$$

where $W_h^V \in \mathbb{R}^{(d+1) \times d_v}$ and Conv1D_h^Q, Conv1D_h^K denote causal 1D convolutions with kernel size k and output dimension d_k.

Explicitly, for the t-th time step, the query and key are

$$q_t^{(h)} = \sum_{i=0}^{k-1} y_{t-i} F_h^Q[i], \quad k_t^{(h)} = \sum_{i=0}^{k-1} y_{t-i} F_h^K[i],$$

where $F_h^Q[i], F_h^K[i] \in \mathbb{R}^{(d+1) \times d_k}$ are the convolutional kernel weights at lag i, and $y_{t-i} \in \mathbb{R}^{1 \times (d+1)}$. This formulation ensures that each query and key explicitly incorporates information from the previous k time steps.

The attention output for head h is then computed as

$$O_h = \text{Attention}(Q_h, K_h, V_h) = \text{softmax}\Big(\frac{Q_h K_h^\top}{\sqrt{d_k}} + M\Big) V_h,$$

where $M \in \mathbb{R}^{t \times t}$ is the causal mask and $O_h \in \mathbb{R}^{t \times d_v}$.

The outputs from all heads are concatenated and linearly projected:

$$O = \text{Concat}(O_1, \dots, O_H)\, W^O, \quad W^O \in \mathbb{R}^{Hd_v \times (d+1)}, \quad O \in \mathbb{R}^{t \times (d+1)}.$$

Finally, each row of O is processed independently by a feedforward network:

$$\mathrm{FFN}(z) = \sigma(zW_1 + b_1)W_2 + b_2,$$

with $z \in \mathbb{R}^{1\times(d+1)}$, $W_1 \in \mathbb{R}^{(d+1)\times d_{\mathrm{ff}}}$, $b_1 \in \mathbb{R}^{d_{\mathrm{ff}}}$, $W_2 \in \mathbb{R}^{d_{\mathrm{ff}}\times(d+1)}$, $b_2 \in \mathbb{R}^{d+1}$, and activation $\sigma(\cdot)$ (ReLU). Applied to all rows, this yields

$$\mathrm{FFN}(O) \in \mathbb{R}^{t\times(d+1)}.$$

Compared to the vanilla Transformer, the key difference is that Q_h and K_h are linear combinations of a local window of past embeddings rather than a pointwise projection, explicitly embedding local temporal patterns before computing attention.

4.2 Breaking the Memory Bottleneck of Transformer

LogSparse / ProbSparse Attention In the standard Transformer, self-attention computes the full pairwise similarity matrix

$$Q_h K_h^\top \in \mathbb{R}^{t\times t},$$

which has quadratic time and space complexity in sequence length t. This limits its applicability to long time series. The *ProbSparse Attention* mechanism (Zhou et al., 2021) alleviates this bottleneck by selecting only the most "informative" queries that contribute significantly to attention, thereby approximating full attention with near-linear complexity.

Let Q_h, K_h, V_h denote the query, key, and value matrices for head h:

$$Q_h, K_h \in \mathbb{R}^{t\times d_k}, \quad V_h \in \mathbb{R}^{t\times d_v}.$$

—

Step 1: Compute sparsity scores for queries.

For each query q_i, ProbSparse defines a *sparsity measurement* that quantifies how concentrated its attention distribution is over all keys. Starting from the scaled dot-product similarity $q_i k_j^\top/\sqrt{d_k}$, define

$$s_i = \ln \sum_{j=1}^{t} \exp\left(\frac{q_i k_j^\top}{\sqrt{d_k}}\right) - \frac{1}{t}\sum_{j=1}^{t} \frac{q_i k_j^\top}{\sqrt{d_k}}.$$

The first term (log-sum-exp) corresponds to the "energy" of query q_i across all keys, while the second term represents its mean similarity. Intuitively, s_i measures the *sharpness* of the attention distribution: a large s_i indicates that q_i attends strongly to a few keys (high sparsity), while a small s_i implies nearly uniform attention. This formulation can be interpreted as the relative entropy (KL divergence) between the attention distribution $p(k_j|q_i)$ and the uniform distribution.

In practice, to reduce computation, the first term is often approximated using the maximum similarity:

$$s_i \approx \|q_i K_h^\top\|_\infty - \frac{1}{t}\sum_{j=1}^{t} q_i k_j^\top,$$

where $\|\cdot\|_\infty$ denotes the maximum value over all keys.

—

Step 2: Select top-u queries.

Select the $u \ll t$ queries with the largest sparsity scores. Denote their indices by $\mathcal{I} \subset \{1,\dots,t\}$, with $|\mathcal{I}| = u$. These "active" queries are used for full attention computation, while the remaining "lazy" queries can be approximated via mean pooling or interpolation.

—

Step 3: Compute attention for selected queries.

For each active query q_i where $i \in \mathcal{I}$, compute the standard scaled dot-product attention:

$$\text{Attention}(q_i, K_h, V_h) = \text{softmax}\left(\frac{q_i K_h^\top}{\sqrt{d_k}} + M_i\right) V_h \in \mathbb{R}^{1\times d_v},$$

where M_i is the causal mask applied to preserve temporal order.

The final attention output for head h is constructed by combining active query outputs with approximations for the skipped ones.

—

Step 4: Concatenate and project.

Outputs from all heads are concatenated and linearly projected as in the standard Transformer:

$$O = \text{Concat}(O_1,\dots,O_H)W^O, \quad W^O \in \mathbb{R}^{Hd_v\times(d+1)}, \quad O \in \mathbb{R}^{t\times(d+1)}.$$

—

Complexity Reduction.

- Standard attention: $O(t^2 d_k)$ - ProbSparse attention: $O(utd_k)$

If $u \sim \log t$, this yields near-linear complexity $O(t \log t\, d_k)$, substantially reducing both memory and computation while preserving dominant attention patterns.

—

Summary.

- Q_h, K_h, V_h are computed as in the vanilla Transformer.
- A sparsity score s_i quantifies how concentrated each query's attention distribution is.
- Only the top-u "active" queries are retained for full attention computation.
- Complexity reduces from quadratic to near-linear $O(t \log t)$.

5 Time Series Transformers

Adapt the architecture discussed in section 3, we attempt to do basic transformer network for multivariate time series. Here, we discuss the changes made to adapt the network to financial time series

5.1 From Learnable Positional Embeddings to Sinusoidal Positional Encodings

In natural language processing (NLP) applications of transformers, it is common to use *learnable positional embeddings*. Each position t within the context window (e.g. token index in a sentence) is assigned a dedicated vector $\mathbf{p}_t \in \mathbb{R}^d$, which is added to the token embedding. This works well for text because only the *ordering* of words within a sentence matters, and the maximum sequence length is fixed by design.

However, in time series forecasting this approach is problematic:

- **Lack of temporal meaning:** In time series, positions correspond to actual time steps. The embedding at position $t = 10$ should encode "10 time steps elapsed," but a learnable embedding is just an arbitrary vector with no relation to elapsed time.
- **Limited generalization:** With a learnable embedding, the model is trained only up to a maximum context length (e.g. 30). At inference, if we wish to roll forward further (e.g. predict 50 steps ahead), no positional vectors exist beyond the training horizon.
- **No periodicity:** Many time series exhibit seasonal cycles (e.g. daily or weekly). A learned embedding does not capture periodic structure unless explicitly memorized from data.

To overcome these issues, the original Transformer paper introduced *sinusoidal positional encodings*, which are *deterministic* and *continuous* functions of position.

Mathematical Formulation

For each position $t \in \{0, 1, \dots, T-1\}$ and embedding dimension $d \in \{0, \dots, D-1\}$, the sinusoidal encoding is defined as:

$$PE_{(t,2i)} = \sin\left(\frac{t}{10000^{2i/D}}\right), \quad PE_{(t,2i+1)} = \cos\left(\frac{t}{10000^{2i/D}}\right),$$

where i indexes over embedding dimensions in steps of 2.

Thus each position t is mapped to a vector $\mathbf{p}_t \in \mathbb{R}^D$ composed of sine and cosine functions with different wavelengths. The denominator $10000^{2i/D}$ ensures that the frequencies are spread geometrically across dimensions, ranging from high-frequency (capturing local order) to low-frequency (capturing long-range position).

Why Sinusoidal Encodings Work Better for Time Series

- **Relative distances are encoded:** Because $\sin(\cdot)$ and $\cos(\cdot)$ are continuous functions, the dot product between two position encodings depends on the relative distance Δt, not just their absolute indices. This is desirable since forecasting often cares about how far back an observation is, not its absolute slot within a fixed window.
- **Extrapolation beyond training horizon:** Unlike learnable embeddings, sinusoidal encodings generalize naturally to positions beyond those seen during training, since the functions are defined for all $t \geq 0$.
- **Periodicity:** The sine and cosine functions naturally capture periodic behaviour. If a series has cycles (daily, weekly, annual), the encoding provides the model with a structured way to detect such repeating patterns.
- **Parameter-free:** Sinusoidal encodings do not introduce additional learnable parameters, reducing the risk of overfitting and making them robust even with limited data.

For these reasons, sinusoidal positional encodings are more suitable than purely learnable embeddings when applying transformers to time series forecasting.

5.2 Activation Function: From ReLU to GELU

A crucial design choice in neural network architectures is the selection of the activation function. In the initial design, the feed-forward network (FFN) within each transformer block employed the Rectified Linear Unit (ReLU), defined as

$$\mathrm{ReLU}(x) = \max(0, x).$$

ReLU has become a standard choice due to its simplicity and effectiveness in preventing vanishing gradients. However, ReLU has a major limitation in the context of financial time series modeling: it discards all negative inputs by setting them to zero. Since asset returns can be both positive and negative, this "hard thresholding" may lead to a systematic loss of information, especially when small negative returns are prevalent.

To address this issue, we switch to the Gaussian Error Linear Unit (GELU) activation which provides a smoother alternative. GELU is defined as

$$\mathrm{GELU}(x) = x\,\Phi(x),$$

where $\Phi(x)$ is the cumulative distribution function (CDF) of the standard normal distribution:

$$\Phi(x) = \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right).$$

This means that the activation is proportional to both the input value x and the probability that a standard Gaussian random variable is less than x. Intuitively, GELU smoothly interpolates between zeroing out very negative inputs and passing through very positive inputs.

Unlike ReLU, which applies a hard cutoff at zero, GELU scales inputs around zero in a continuous manner, retaining partial information from small negative values rather than discarding them completely. This property is particularly beneficial for return series, where sign and magnitude carry valuable predictive information.

Empirically, GELU has been shown to outperform ReLU in transformer-based architectures such as BERT and GPT, improving gradient flow and training stability in deep networks. By adopting GELU, the model benefits from a smoother nonlinearity that better accommodates the distributional characteristics of financial returns.