More On Neural Networks

Contents

Dropout

Dropout is a regularization technique used in neural networks to prevent overfitting. It works by randomly "dropping out" (deactivating) a subset of neurons during each training iteration, making the network less sensitive to specific neurons and reducing the likelihood of overfitting.

Dropout (training)

During training, dropout randomly deactivates a fraction of neurons in the layer it's applied to. The neurons are "dropped out" by setting their activations to zero.

The probability of a neuron being dropped out is defined by a hyperparameter **p** (dropout rate), where $0 \leq p < 1$. For example, if $p=0.5$, then each neuron has a 50% chance of being dropped out during training.

The weights of the active neurons are adjusted as usual during backpropagation. The dropped-out neurons do not contribute to forward or backward passes in that iteration.

$\mathbf{h} = \mathbf{r} \odot \mathbf{x}$ with $r_i \sim \text{Bernulli}(1-p)$ and $\odot$ denoting element-wise multiplication.

Dropout (inference)

During inference (testing or validation), no neurons are dropped out. Instead, the activations are scaled down by the dropout rate p (typically by multiplying by 1−p) to account for the fact that fewer neurons were active during training.

$\mathbf{h} = (1\text{-}p)\mathbf{x}$

Prevents Overfitting and Improves generalization

By randomly deactivating neurons, dropout forces the network to learn more robust features that do not depend on the presence of specific neurons.

(Not always effective)

But How it is Really Implemented?

In the **inverse dropout** formulation, the scaling is applied during training instead of inference. This ensures that the expected value of the activations remains consistent between training and inference, and there is no need to scale during inference.

The equation for a single dropout layer with inverse dropout is:

$$\mathbf{h} = (\mathbf{r} \odot \mathbf{x}) / (1 - p)$$

Vanishing Gradient

The **vanishing gradient problem** occurs in deep neural networks, particularly in very deep networks with many layers. During backpropagation, gradients (the partial derivatives of the loss with respect to each parameter) become increasingly small as they propagate backwards from the output layer to the earlier layers.

- Slow Convergence
- Poor Training
- Poor Performance

Residual Connections

Residual connections (or **skip connections**) were introduced in **Residual Networks (ResNets)** to address this problem. A residual connection "skips" one or more layers by adding the input of a layer directly to its output, which can be mathematically represented as:

$\mathbf{h} = F(\mathbf{x}) + \mathbf{x}$

Each layer adds something (i.e. a residual) to the previous value, rather than producing an entirely new value.

Backpropagation

We can string together a bunch of residual units.

What happens if we set the parameters such that f(x) = 0?

- Then it passes x straight through unmodified!
- This means it's easy for the network to represent the identity function.

Backprop:

$\nabla_x \mathbf{h} = \nabla_x (F(\mathbf{x}) + \mathbf{x}) = \partial F / \partial x + I$ ($\nabla_x x = I$ the identity matrix)

This means the derivatives don't vanish.

Deep Residual Networks

Deep Residual Networks (ResNets) consist of many layers of residual units.

For vision tasks, the F functions are usually 2 or 3 layer conv nets.

For a regular convnet, performance declines with depth, but for a ResNet, it keeps improving.

ImageNet

- A 152-layer ResNet achieved 4.49% top-5 error on Image Net. An ensemble of them achieved 3.57%.
- Previous state-of-the-art: 6.6% (GoogLeNet)
- Humans: 5.1%

They were able to train ResNets with more than 1000 layers, but classification performance leveled off by 150.

What are all these layers doing? We don’t have a clear answer…

More results here:
https://paperswithcode.com/sota/image-classification-on-imagenet

Standard Scaling

Given a dataset X with dimensions (n, d), where n is the number of samples and d is the number of features, it is common in machine learning to preprocess the data such that each feature (column) has zero mean and unit variance.

$\mathbf{X'} = (\mathbf{X} - \boldsymbol{\mu}) / \mathrm{sqrt}(\boldsymbol{\sigma}^2)$

where:

μ is the mean of each feature/column, calculated as: $\mu_j = (1/n) \sum_{(i=1 \text{ to } n)} X_{ij}$

σ² is the variance of each feature, calculated as: $\sigma^2_j = (1/n) \sum_{(i=1 \text{ to } n)} (X_{ij} - \mu_j)^2$

Batch Normalization

Batch Normalization (BN), extends the concept of data preprocessing by normalizing the **outputs of each layer** or **block** in a neural network. The goal is to learn an optimal mean and variance for each unit of the network’s layers during training.

Not straightforward! The mean and variance of the layer's output can change during the optimization process.

Instead of recalculating the statistics over the whole dataset, BN approximates the mean and variance by using the data in a **mini-batch**.

BN Training

Given an output H with dimensions (b, d), where b is the batch size. We compute mean and variance and standardize it:

$\mathbf{H'} = (\mathbf{H} - \boldsymbol{\mu}) / \ (\mathrm{sqrt}(\boldsymbol{\sigma}^2) + \varepsilon$, where $\varepsilon > 0$ is a small coefficient to avoid division by 0.

The final output of the batch normalization layer BN(**H**) sets a new mean and variance for each column:

$\mathrm{BN}(H) = \alpha_j H'_{i,j} + \beta_j$ for j=1 to d.

The 2d values α_j and β_j are trained via gradient descent.

BN Inference

During inference since it is undesirable for the output of an input to depend on the mini-batch it is part of. Two common solutions are:

Post-Training Statistics Calculation
After training, compute the mean and variance by running the trained model on the entire dataset, and fix the values to these computed statistics.

Moving Average of Statistics
During training, maintain a moving average of the estimated means and variances for each feature. At inference time, use the final moving averages for normalization, ensuring that the model's behavior is consistent with what it has learned during training.

More on BN

What if we have the output of a convolution (b,h,w,c)?

Batch Normalization (BN) works in the same way as before, but with a slight modification: the **mean** and **variance** are computed **per channel**. This means that for each channel c, the normalization is applied independently across the spatial dimensions h and w, as well as the batch dimension b.

Challenges

Mini-Batch Dependencies

BN creates dependencies between the elements within a mini-batch, which can limit its effectiveness in scenarios such as distributed optimization or contrastive self-supervised learning.

High Variance with Small Batch Sizes

When using small batch sizes, the variance in the computed mean and variance estimates can become excessively high. This issue is particularly problematic in large models that require smaller batches to fit within memory constraints, leading to unstable training.

Layer Normalization

Layer Normalization normalizes the inputs to a layer **across the features** (instead of the mini-batch). This technique is commonly used in **forecasting neural networks working with time series** and **transformers**, where the batch size can vary or is often set to 1.

Just change the axis!

The 2b values α_i and β_i are trained via gradient descent.

Data Augmentation

Technique used to effectively increase the size of the training dataset by applying random transformations to the input data.

1) **Sample a mini-batch** of examples from the dataset.
2) For each example, apply one or more **random transformations**, such as flipping, cropping, rotating, etc.
3) **Train** the model on the transformed mini-batch.

Benefits of Data Augmentation: Prevents overfitting and enhances model **robustness** to small variations in input data, improving generalization to new, unseen examples.

List of Transformations

Geometric Transformations:

- **Flipping**: Horizontal/vertical flip
- **Rotation**: Random angle rotation
- **Translation**: Random shifts in x/y axes
- **Scaling**: Resize while maintaining aspect ratio
- **Cropping**: Random region cropping
- **Zooming**: Zoom in or out
- **Affine**: Combine translation, scaling, rotation, shear

Color and Lighting Adjustments:

- **Brightness/Contrast/Saturation/Hue**: Random adjustments
- **Color Jittering**: Modify brightness, contrast, saturation, and hue together
- **Grayscale**: Convert to grayscale with probability

Noise and Distortion:

- **Gaussian Noise**: Add random noise
- **Salt-and-Pepper Noise**: Random pixel value change
- **Elastic Deformations**: Apply spatial distortions

Cutout/Masking:

- **Cutout**: Randomly mask rectangular regions
- **Random Erasing**: Occlude parts with color/texture

Combination Techniques:

- **Mixup**: Weighted sum of two images
- **CutMix**: Paste a patch from one image into another

1-D Convolution

Commonly used for **time series** data where information is ordered in a sequence. The goal is to extract local features or patterns that evolve over time.

Consider a time series of n steps, $x_0, x_1, \ldots, x_{n-1}$, where each step has c features.

Represent the time series as a matrix X (n,c), where each row corresponds to a time step, and each column represents a feature.

A 1D convolution with a receptive field of size 2k is defined as:

$$H_i = \phi\left(\sum_{j=-k}^{k} \sum_{z=1}^{c} W_{j+k,z} \cdot X_{i+j,z}\right)$$

What do the do?

Local Pattern Detection

1D convolutions capture local dependencies in time series, detecting trends or repeated patterns over time.

Parameter Sharing

The same filter is applied across all time steps, reducing the number of parameters and improving generalization.

Translation Invariance

Helps in identifying features that are present at different time steps, making it robust to shifts in the time domain.

Of course… same as 2D convolution.

Common Applications

Forecasting: Predict future values in time series, e.g., stock prices

Anomaly Detection: Detect unusual patterns or outliers in sensor readings or other time-dependent data.

Classification: Classify sequences based on temporal patterns, e.g., activity recognition in wearable sensor data.

Signal Processing: Apply to filtering or denoising time-series signals.

Causal Convolution

Causal Convolution ensures that the output at each time step i only depends on the current and previous time steps, not future ones. This is important for **time series forecasting** or any model where data at future time steps shouldn't influence the current prediction.

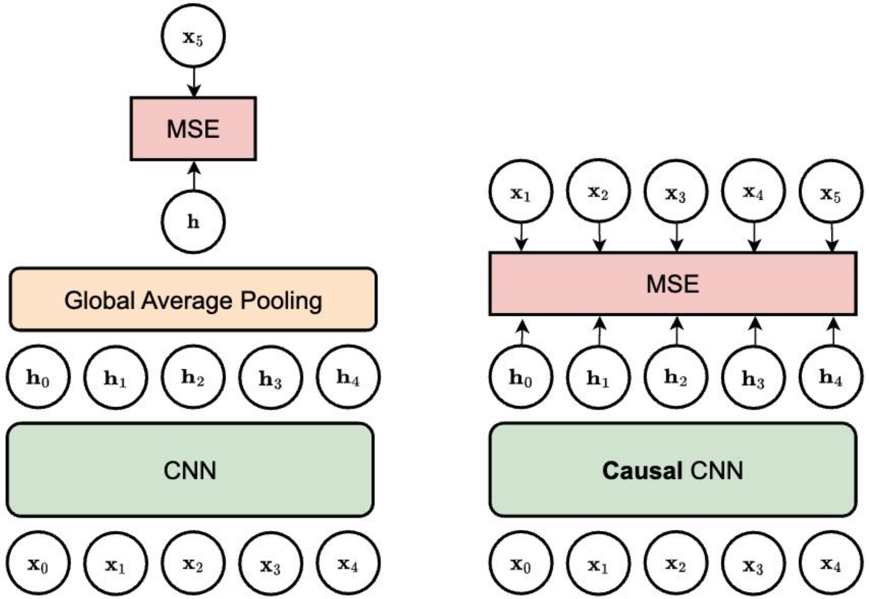
$$H_i = \phi\left(\sum_{j=0}^{k}\sum_{z=1}^{c} W_{j,z} \cdot X_{i-j,z}\right)$$

Causal Model

For time-series, a common task is forecasting, i.e., predicting the next step in the time-series. With a causal model, we have two options:

- Pool the output representation H over n, and apply a regressor head to predict x_n (also valid for non-causal models).
- Define a shifted target Y = $[x_1, \dots, x_n]$, and train the model such that $H_i \approx H_{i+1}$, i.e., at each time step the network predicts the next one. This is only possible with a causal model, otherwise information would ‘leak’ from the input.

Some Pictures



$\mathbf{x}_5$
MSE
$\mathbf{h}$
Global Average Pooling
$\mathbf{h}_0$ $\mathbf{h}_1$ $\mathbf{h}_2$ $\mathbf{h}_3$ $\mathbf{h}_4$
CNN
$\mathbf{x}_0$ $\mathbf{x}_1$ $\mathbf{x}_2$ $\mathbf{x}_3$ $\mathbf{x}_4$
$\mathbf{x}_1$ $\mathbf{x}_2$ $\mathbf{x}_3$ $\mathbf{x}_4$ $\mathbf{x}_5$
MSE
$\mathbf{h}_0$ $\mathbf{h}_1$ $\mathbf{h}_2$ $\mathbf{h}_3$ $\mathbf{h}_4$
Causal CNN
$\mathbf{x}_0$ $\mathbf{x}_1$ $\mathbf{x}_2$ $\mathbf{x}_3$ $\mathbf{x}_4$

Image from Neural Networks for Data Science Applications (Prof. Scardapane)

Autoregressive Generation

Models trained to predict the next step in a sequence can be used for **autoregressive generation**. The process is as follows:

1) Start with an initial input sequence as a prompt.
2) The model predicts the next value in the sequence.
3) Append this prediction to the input and use the updated sequence to forecast the next step. Repeat!

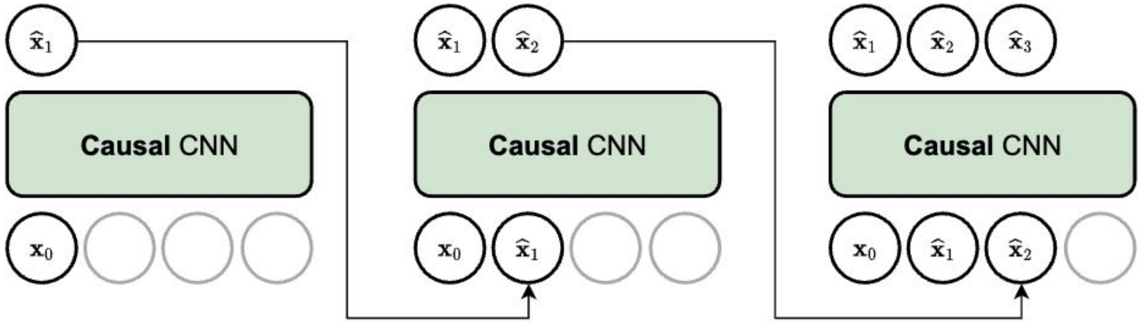


Image from Neural Networks for Data Science Applications (Prof. Scardapane)

The Famous Self-Attention

1D Convolution and **Self-Attention** are both techniques used in sequence modeling, particularly for tasks like time series analysis, natural language processing, and other temporal data.

In 1D convolution, a filter of fixed size slides over the input sequence, capturing local patterns by aggregating information from a limited receptive field.

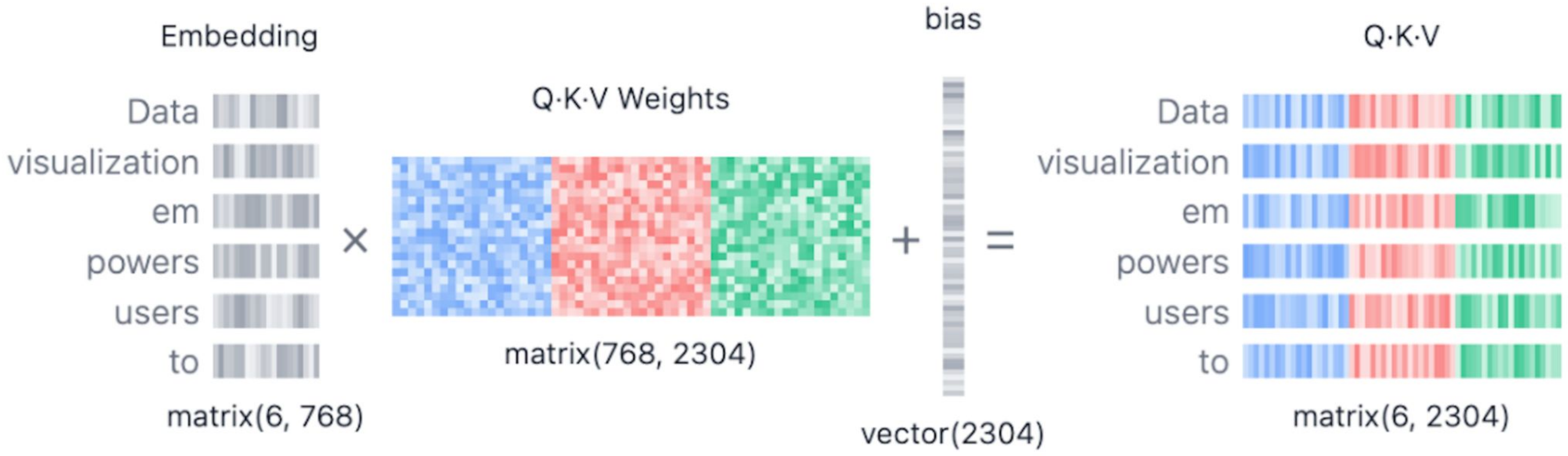
Self-attention computes pairwise interactions between all elements in the sequence. For each element, it calculates attention scores with every other element, enabling it to learn dependencies across the entire sequence regardless of their distance.

The output is a weighted sum of all elements, where the weights are learned attention scores indicating the importance of each element in relation to the others.

1-D Conv vs Self-Attention

1D Conv: Local, shared weights, efficient.
Self-Attention: Global, weights for each pair of inputs, quadratic complexity.

Query, Key, and Value Matrices



$$\sum_{d=1}^{768} E_{id} \cdot W_{dj} + b_j = QKV_{ij}$$

Query, Key, and Value Matrices

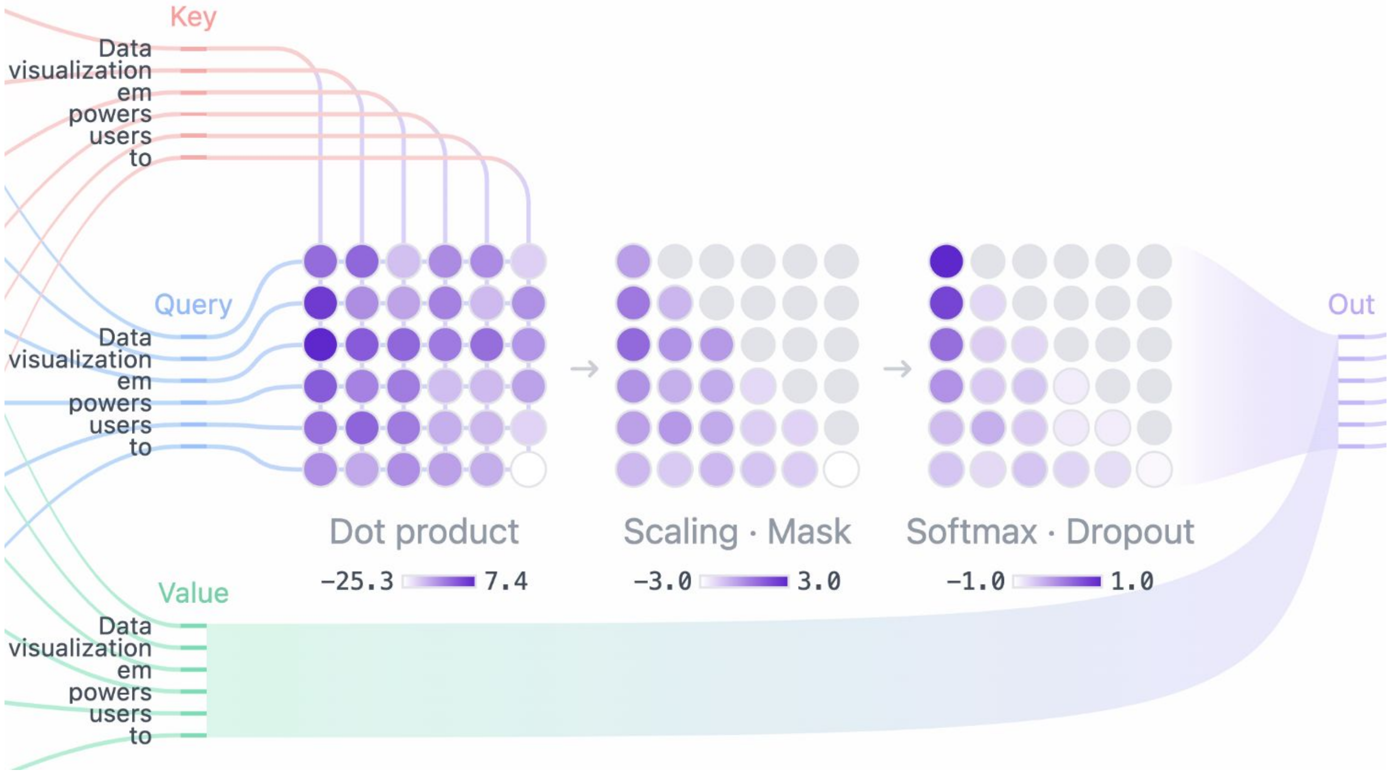
Each token's embedding vector is transformed into three vectors: **Query (Q)**, **Key (K)**, and **Value (V)**. These are obtained by multiplying the embedding with learned weight matrices for Q, K, and V.

Analogy with a Web Search:

- **Query (Q)**: The search term you type—what you're looking for.
- **Key (K)**: The titles of web pages—potential matches for the search.
- **Value (V)**: The content of the web pages—what you actually want to read.

The model uses these Q, K, and V vectors to compute **attention scores**, determining how much focus each token should receive when making predictions.

Masked Self-Attention



Masked Self-Attention

Attention Score: The **dot product** of the Query and Key vectors measures the similarity between each query and key, forming a matrix that captures the relationships among all input tokens.

Masking: A mask is applied to the upper triangle of this matrix to block future tokens, setting their scores to negative infinity. This prevents the model from accessing information from future steps when predicting the next token.

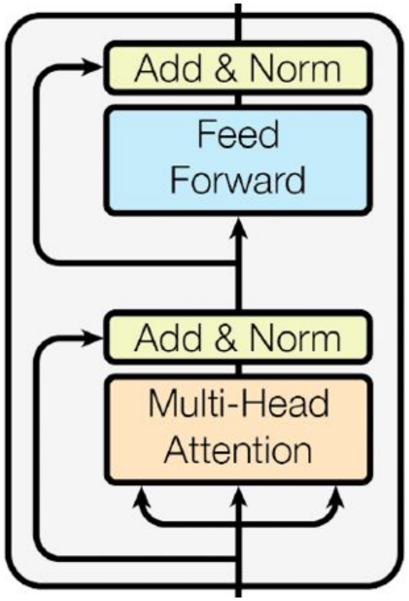
Softmax: The masked scores are passed through a **softmax** function, converting them into probabilities. Each row sums to one, indicating the relevance of each preceding token based on its alignment score.

Multi-Head and MLPs

The model uses the masked self-attention scores and multiplies them with the Value matrix to get the final output of the self-attention mechanism. GPT-2 has 12 self-attention heads, each capturing different relationships between tokens. The outputs of these heads are concatenated and passed through a linear projection.

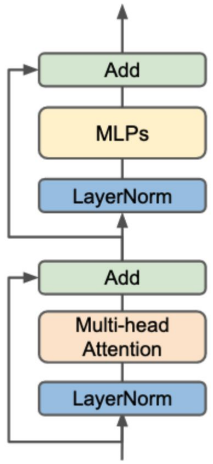
After the multiple heads of self-attention capture the diverse relationships between the input tokens, the concatenated outputs are passed through the Multilayer Perceptron (MLP) layer to enhance the model's representational capacity.

Transformer Block



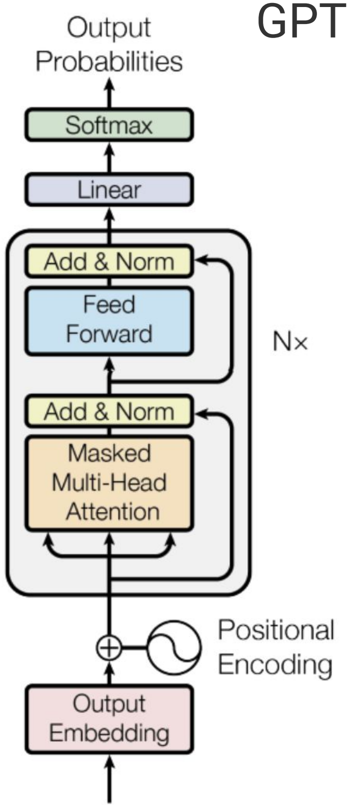
Add is just a Residual Connection.

Norm is a Layer Normalization.



Layernorm now is applied before MHA and MLP

GPT



Building Text Embeddings

How do we represent an input text?

Text input is divided into smaller units called tokens, which can be words or subwords. These tokens are converted into numerical vectors called embeddings, which capture the semantic meaning of words.

Word encoders have several issues: Need to detect boundary of words (“O’Neill”, “don’t”), Word-level tokenization treats different forms of the same word as separate types (“open”, “opened”, “opens”, “opening”)

Char encoders (a->82) reduce complexity but are almost impossible to use.

SubWord Encoders

Relies on a simple algorithm called byte pair encoding (Gage, 1994)

How does it work?

Form base vocabulary (all characters that occur in the training data)

word	frequency
hug	10
pug	5
pun	12
bun	4
hugs	5

Base vocab: b, g, h, n, p, s, u

Byte Pair Encoding

Now, count up the frequency of each character pair in the data, and choose the one that occurs most frequently

word	frequency
h+u+g	10
p+u+g	5
p+u+n	12
b+u+n	4
h+u+g+s	5

character pair	frequency
ug	20
pu	17
un	16
hu	15
gs	5

Byte Pair Encoding

Choose the most common pair (ug) and then merge the characters together into one symbol. Add this new symbol to the vocabulary. Then, re-tokenize the data

word	frequency
h+*ug*	10
p+*ug*	5
p+u+n	12
b+u+n	4
h+*ug*+s	5

character pair	frequency
un	16
h+ug	15
pu	12
p+ug	5
ug+s	5

Byte Pair Encoding

Stop after a fixed number of steps

word	frequency
hug	10
p+*ug*	5
p+*un*	12
b+*un*	4
hug + s	5

GPT-2 tokenizes text by first converting it into bytes (covering all characters with a 256-size base vocabulary) and then applies Byte Pair Encoding (BPE) on top to merge frequently occurring byte sequences into subword tokens, with specific rules to control the merging process.

new vocab: b, g, h, n, p, s, u, ug, un, hug

Positional Encoding

Lack of Sequential Awareness

Transformer models process all tokens simultaneously and do not inherently consider their order. Without positional encodings, the model would treat "cat sat on the mat" and "mat sat on the cat" as identical, ignoring the sequence information.

Adding Positional Information

Positional encodings inject order into the model by embedding position-specific information into the token vectors, enabling the transformer to differentiate tokens based on their position in the sequence.

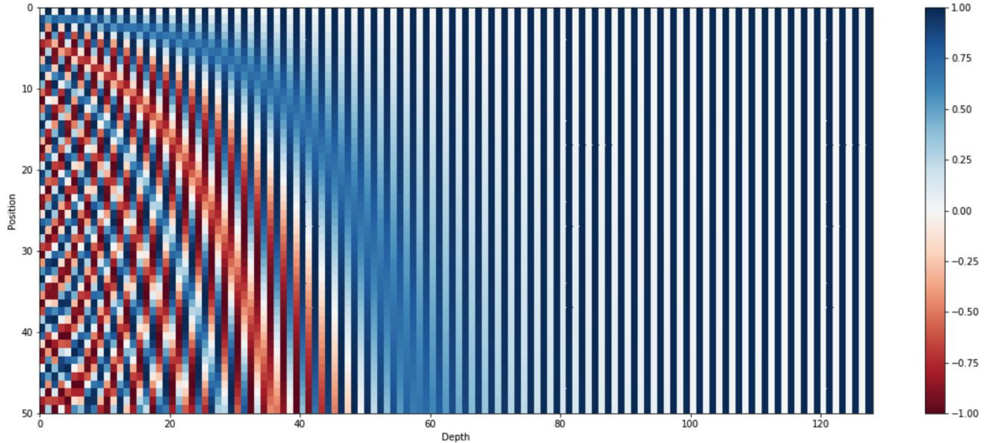
Sinusoidal Embeddings

Let t be the desired position in an input sentence, $\overrightarrow{p_t} \in \mathbb{R}^d$ be its corresponding encoding, and d be the encoding dimension (where $d \equiv_2 0$) Then $f : \mathbb{N} \rightarrow \mathbb{R}^d$ will be the function that produces the output vector $\overrightarrow{p_t}$ and it is defined as follows:

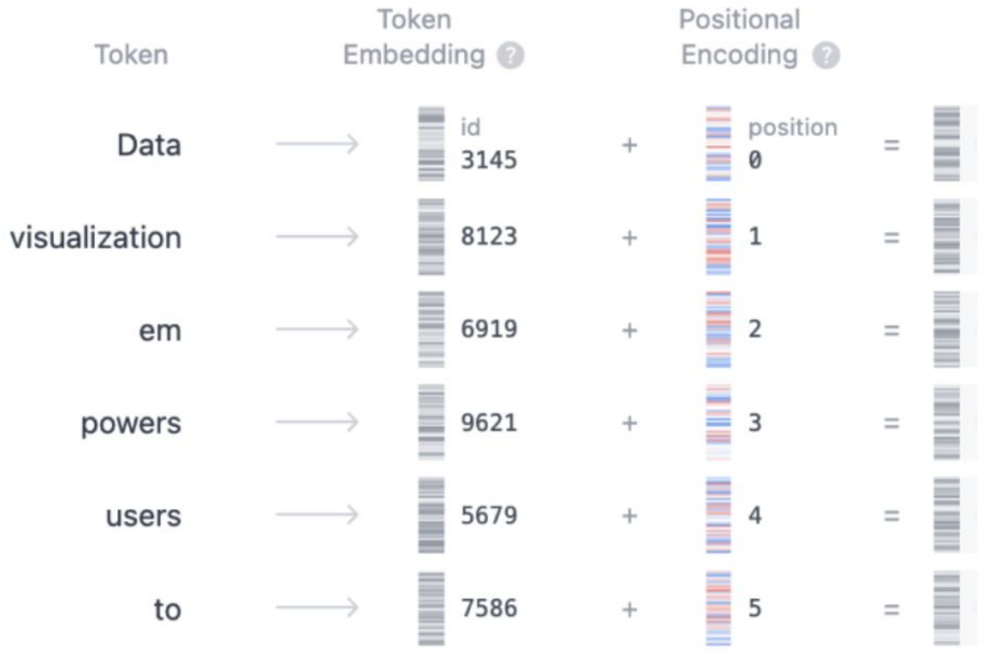
$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases}$$

where

$$\omega_k = \frac{1}{10000^{2k/d}}$$



GPT Input



Token
Token Embedding
Positional Encoding
Data
visualization
em
powers
users
to
id
3145
8123
6919
9621
5679
7586
position
0
1
2
3
4
5

Relative Positional Embeddings

The attention mechanism considers the **relative distance** $i−j$ between tokens instead of their absolute positions.

For instance, in **Attention with Linear Biases (ALiBi)**, trainable biases b_{ij} are added based on the relative positions, allowing the model to learn distance-based relationships directly in the attention scores.

Visualization of ALiBi

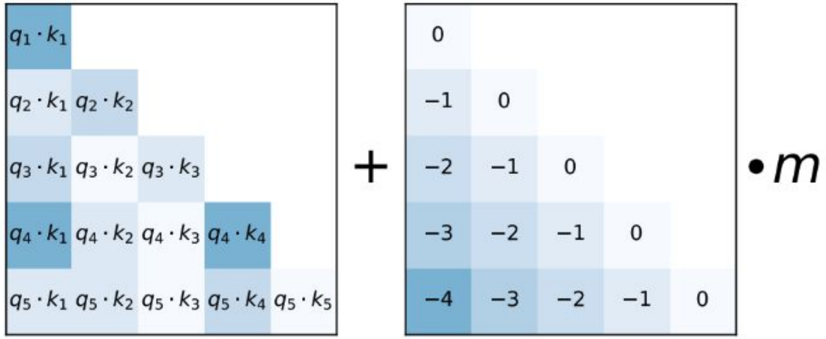

$q_1 \cdot k_1$
$q_2 \cdot k_1$ $q_2 \cdot k_2$
$q_3 \cdot k_1$ $q_3 \cdot k_2$ $q_3 \cdot k_3$
$q_4 \cdot k_1$ $q_4 \cdot k_2$ $q_4 \cdot k_3$ $q_4 \cdot k_4$
$q_5 \cdot k_1$ $q_5 \cdot k_2$ $q_5 \cdot k_3$ $q_5 \cdot k_4$ $q_5 \cdot k_5$
+
0
−1 0
−2 −1 0
−3 −2 −1 0
−4 −3 −2 −1 0
$\cdot m$

Figure 3: When computing attention scores for each head, our linearly biased attention method, ALiBi, adds a constant bias (right) to each attention score ($\mathbf{q}_i \cdot \mathbf{k}_j$, left). As in the unmodified attention sublayer, the softmax function is then applied to these scores, and the rest of the computation is unmodified. **m is a head-specific scalar** that is set and not learned throughout training. We show that our method for setting m values generalizes to multiple text domains, models and training compute budgets. When using ALiBi, we do *not* add positional embeddings at the bottom of the network.

Transformer outputs

After the input passes through all Transformer blocks, the resulting output is fed into a final linear layer that projects it into a **50,257-dimensional space**—matching the vocabulary size. Each value in this output, called a **logit**, represents the model's raw prediction score for each token in the vocabulary. To predict the next word, we apply the **softmax** function to these logits, converting them into a probability distribution where each token's probability reflects its likelihood of being the next word. This distribution can then be used to select the most probable token or sample one based on its likelihood.

Temperature

To generate the next token, we sample from the probability distribution, adjusting the **temperature** hyperparameter to control the randomness of the output:

- **Temperature = 1**: No effect on logits; the output remains unchanged.
- **Temperature < 1**: The distribution sharpens, making the model more confident and outputs more predictable.
- **Temperature > 1**: The distribution softens, increasing randomness and allowing for more varied, creative outputs.

By tuning the temperature, we can balance between deterministic and diverse text generation.

References

Interactive Transformer: https://poloclub.github.io/transformer-explainer/

Tokenization: https://huggingface.co/docs/transformers/tokenizer_summary

PE: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

Interactive Tokenizer: https://platform.openai.com/tokenizer