LLMs

What is a language model?

A language model is a probabilistic model that assign probabilities to sequence of words.
In practice, a language model allows us to compute the following:

P ["China" | "Shanghai is a city in"]

Next Token **Prompt**

We usually train a neural network to predict these probabilities. A neural network trained on a large corpora of text is known as a Large Language Model (LLM).

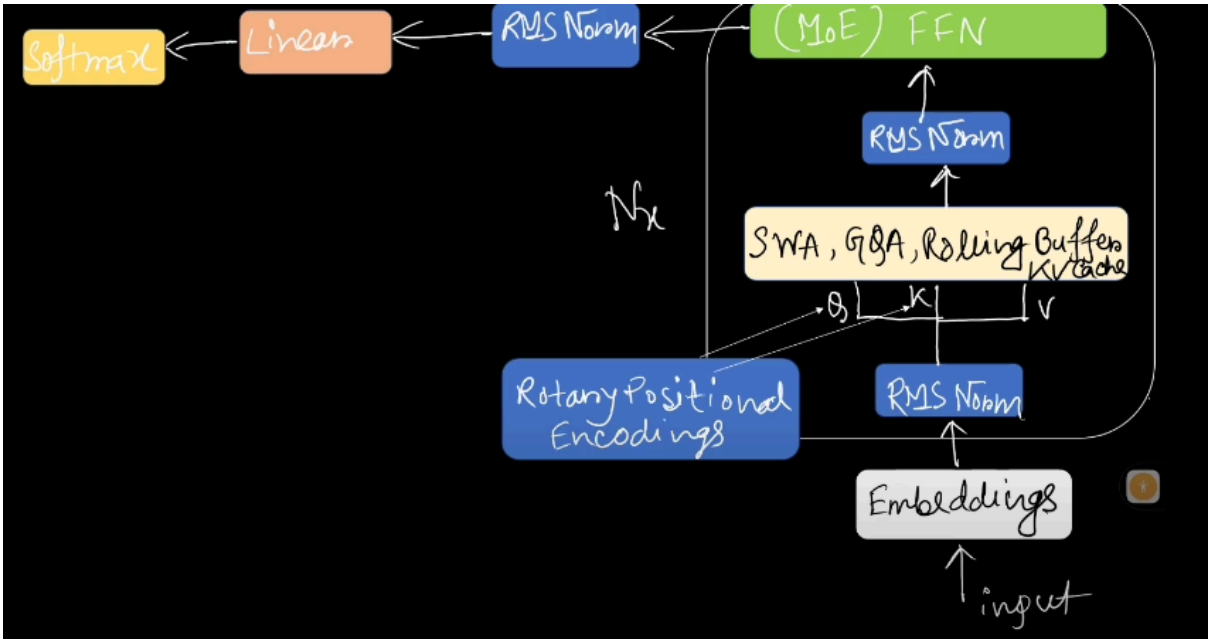
Self-Attention mechanism: causal mask

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

	[SOS]	Before	my	bed	lies	a	pool	of	moon	bright
[SOS]	**5.45**	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞
Before	4.28	**2.46**	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞
my	8.17	3.56	**5.54**	-∞	-∞	-∞	-∞	-∞	-∞	-∞
bed	6.71	4.13	6.76	**0.79**	-∞	-∞	-∞	-∞	-∞	-∞
lies	5.43	7.59	3.91	6.14	**9.03**	-∞	-∞	-∞	-∞	-∞
a	4.42	4.35	7.55	3.14	1.35	**7.57**	-∞	-∞	-∞	-∞
pool	8.36	6.00	4.56	0.52	3.13	6.78	**9.00**	-∞	-∞	-∞
of	2.21	3.72	4.16	6.30	0.66	6.14	7.46	**6.77**	-∞	-∞
moon	4.08	6.22	5.00	4.20	5.72	5.35	7.46	3.55	**4.70**	-∞
bright	6.43	8.88	6.17	3.65	4.54	5.22	5.51	5.55	0.64	**1.38**

	[SOS]	Before	my	bed	lies	a	pool	of	moon	bright
[SOS]	**1.00**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Before	0.86	**0.14**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
my	0.92	0.01	**0.07**	0.00	0.00	0.00	0.00	0.00	0.00	0.00
bed	0.47	0.04	0.49	**0.00**	0.00	0.00	0.00	0.00	0.00	0.00
lies	0.02	0.18	0.00	0.04	**0.75**	0.00	0.00	0.00	0.00	0.00
a	0.02	0.02	0.47	0.01	0.00	**0.48**	0.00	0.00	0.00	0.00
pool	0.31	0.03	0.01	0.00	0.00	0.06	**0.59**	0.00	0.00	0.00
of	0.00	0.01	0.02	0.15	0.00	0.12	0.47	**0.23**	0.00	0.00
moon	0.02	0.16	0.05	0.02	0.10	0.07	0.55	0.01	**0.03**	0.00
bright	0.07	0.71	0.05	0.03	0.01	0.02	0.03	0.03	0.02	**0.03**

Mistral



Mistral 7B is a decoder only model which means that it resembles the decoder block of the transformer architecture. Most language models today are decoder only model since they are designed for text generation which doesn't need bidirectional processing. Mistral 7B uses embedding of size 4096 which means that each token is represented by a 4096 dimensional vector. Context length of Mistral 7B is 8192 which means that it can consider up to 8192 tokens of text when predicting the next token in a sequence.

Rotary Positional Embeddings

What's the difference between the absolute positional encodings and the relative ones?

- Absolute positional encodings are fixed vectors that are added to the embedding of a token to represent its absolute position in the sentence. So, it deals with **one token at a time**. You can think of it as the pair (latitude, longitude) on a map: each point on earth will have a unique pair.
- Relative positional encodings, on the other hand, deals with two tokens at a time and it is involved when we calculate the attention: since the attention mechanism captures the "intensity" of how much two words are related two each other, relative positional encodings tells the attention mechanism the **distance** between the two words involved in it. So, given **two tokens**, we create a vector that represents their distance.

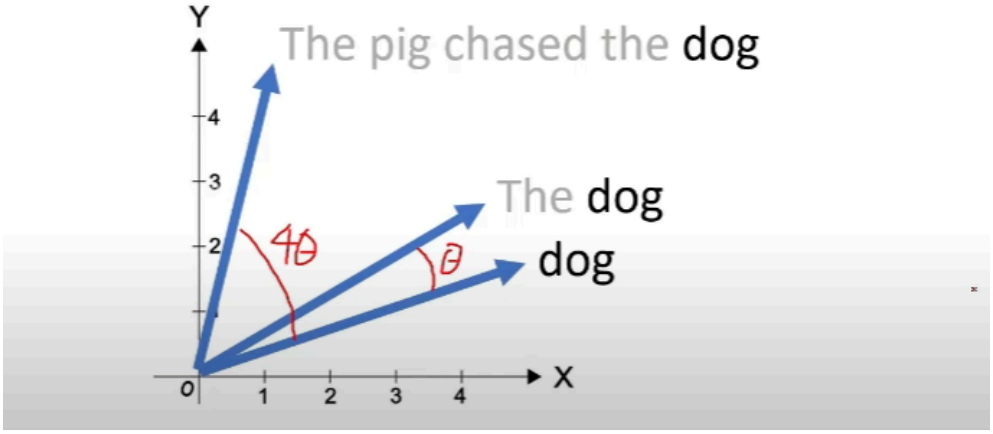
Absolute Positional Encodings
From *"Attention is all you need"*

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d_z}}$$

Relative Positional Encodings
From *"Self-Attention with relative positional representations"*

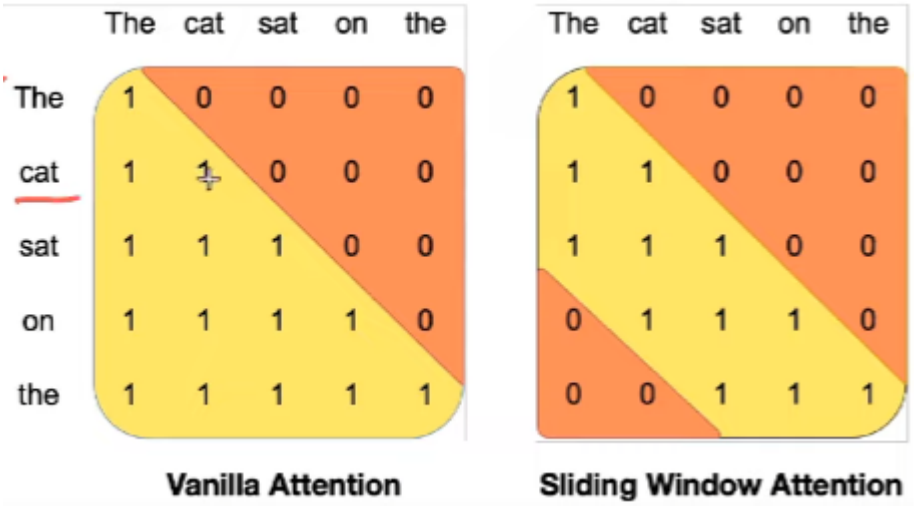
$$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}}$$

Rotary Positional Embeddings



Can encode relative and absolute positional information and can be faster unlike relative embeddings which are slow and sinusoidal embedding which doesn't encode relative information.

Sliding Window Attention



In the normal self attention mechanism, you basically have every token attend to every other token in the sentence that comes before it. But, Mistral applies a sliding window of some size w that doesn't let a token attend to tokens beyond the window size. As you can see in the diagram below, a window size of 3 has been maintained along with the causal mask and this significantly helps in faster training and inference since you have to do a lot less dot product computations. However this might lead to a bit of degradation in output quality since we may not be able to capture the complete local context. But, as we know language models are comprised of multiple transformer block stacked upon each other meaning we have n numbers of transformer block that repeat one after other which is 32 in the case of Mistral 7B. This allows tokens to indirectly find relationship with other tokens that may not fall within the direct window size of the sliding window attention mechanism. For instance, take a look at the sliding window attention image below where the word 'on' doesn't take into consideration the first word 'The' but due to the nature of stacked transformers blocks, the

word ‘on’ indirectly attends to the word ‘The’. Hence, even though the sliding window attention mechanism restricts direct attention to a local neighbourhood around each token, the multiple layers of transformer blocks enable information to propagate across the sequence.

Rolling Buffer Cache

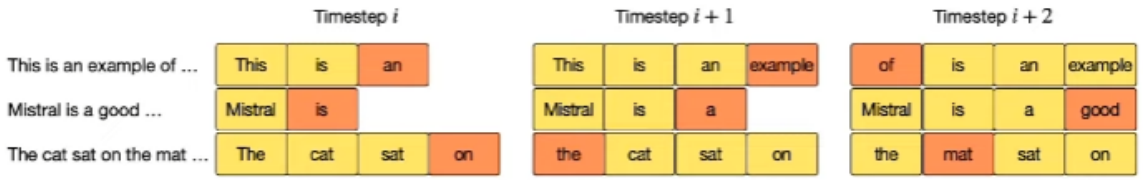
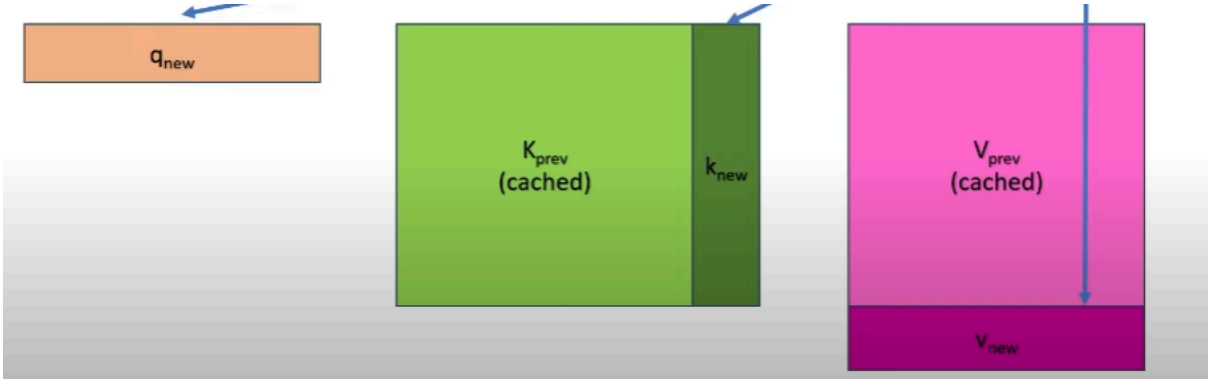


Figure 2: Rolling buffer cache. The cache has a fixed size of $W = 4$. Keys and values for position i are stored in position $i \bmod W$ of the cache. When the position i is larger than W, past values in the cache are overwritten. The hidden state corresponding to the latest generated tokens are colored in orange.

Since Mistral is a decoder only model, it is trained on the next token prediction task. How inference is performed is that it starts with a special token known as start token as input. From there, it generates the first word based on the start token. Then, using both the start token and the first word as context, the model generates the second word, and this process continues iteratively until another special token called end token is encountered. So, at each time step we predict the next token, concatenate it with the input and repeat the process. This iterative process involves a lot of redundant computation of the same token that are already computed in previous time step. Hence, a method known as key value (KV) cache is employed in Mistral to optimize this process where only the key and value vectors are cached while the query vector is updated at each step. This allows the model to reuse the key and value vectors across multiple steps reducing the redundant computations associated with generating each token and thereby making inference faster. But, since Mistral employs the sliding window technique in the attention block, we do not even need to perform computation for tokens that do not fall under the window size. Hence, the rolling buffer cache is used that basically limits the size of cache to the size of sliding window. How it works is that the keys and values for the timestep i are stored in position(i mod W) of the cache. So, when the position i is larger than size of window(w) , past values in the cache are overwritten and the size of the cache stops increasing.



Prefill and Chunking

We always know the prompt before hand , so can we ***prefill*** the KV with the prompts to optimize the performence and generate the future tokens?

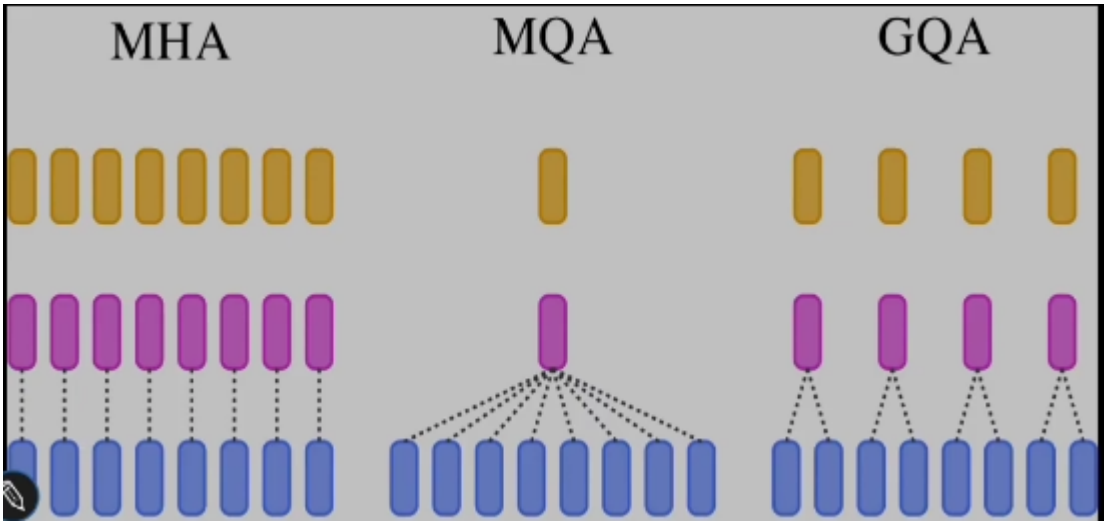
But what if the prompt is very large ? (like 6000/8000 token in case of RAG)

So can we strike a balance between filling the KV Cache with one token at a time or loading the KV Cache with the full prompt ?

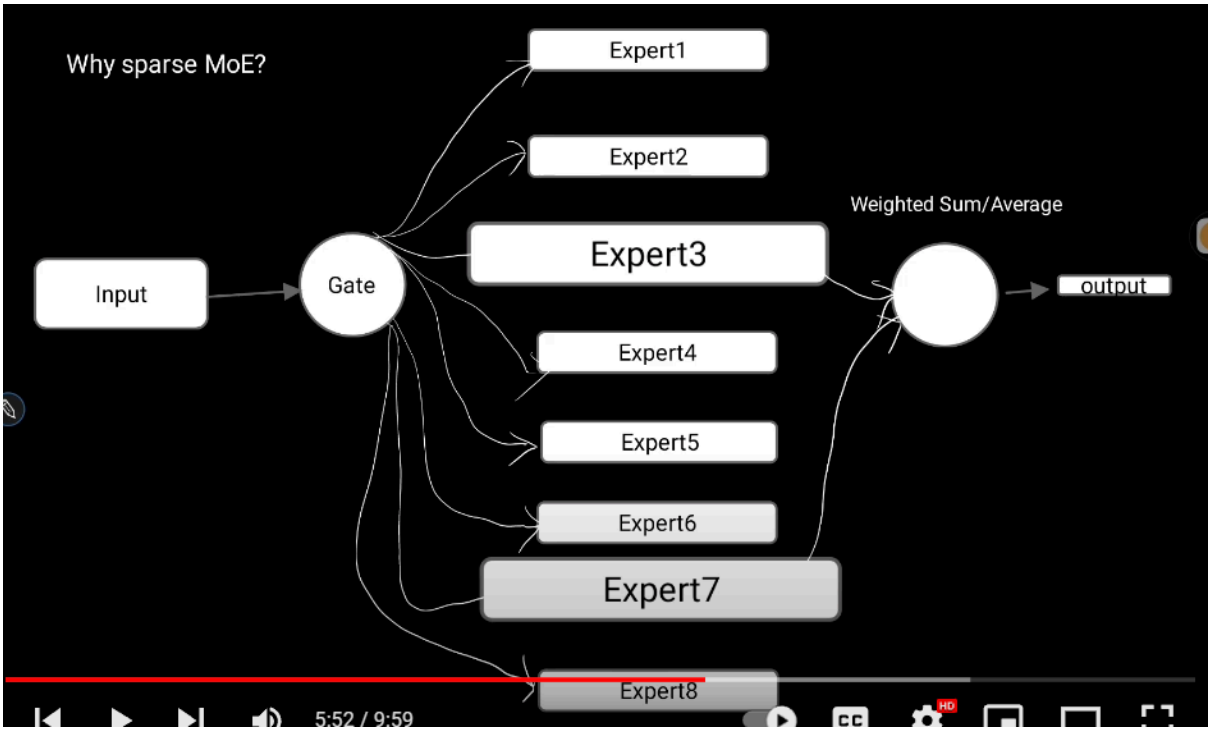
Answer: ***Chunking*** the prompt with the Window size and then prefill the KV Cache

Flash Attention

Grouped query attention



Mixture of experts



Each expert(7B) is trained on a subset of data so that each expert is specialized on certain tasks.

Gating layer is a linear layes which was trained along with the experts and rest of the models.

For each input token embeddings , gate produces 8 logits , which also used to select 2 experts with highest logit values.

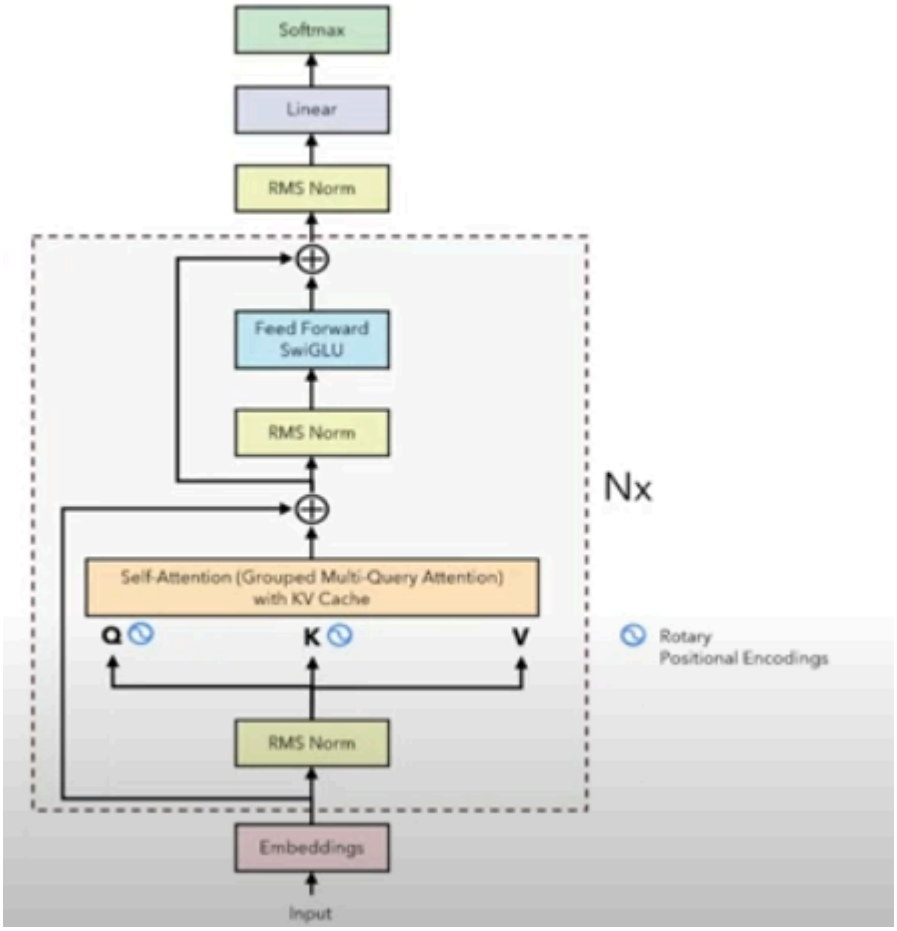
Logits are then normalized using softmax function to produce the weights.

These weights are then used to produce the final output by taking weighted sum or average.

Sparse MoE allows us to increase the parameters of the model , without increase the computation/inference time as intermediate computations will only happen so selected Experts.

LLAMA

Decoder based



RMS Norm

Layer normalization (LayerNorm) has been successfully applied to various deep neural networks to help stabilize training and boost model convergence because of its capability in handling re-centering and re-scaling of both inputs and weight matrix. However, the computational overhead introduced by LayerNorm makes these improvements expensive and significantly slows the underlying network, e.g. RNN in particular.

A well-known explanation of the success of LayerNorm is its re-centering and re-scaling invariance property. The former enables the model to be insensitive to shift noises on both inputs and weights, and the latter keeps the output representations intact when both inputs and weights are randomly scaled. In this paper, we hypothesize that the re-scaling invariance is the reason for success of LayerNorm, rather than re-centering invariance. We propose RMSNorm which only focuses on re-scaling invariance and regularizes the summed inputs simply according to the root mean square (RMS) statistic:

$$\bar{a}_i = \frac{a_i}{\mathrm{RMS}(\mathbf{a})} g_i, \quad \text{where } \mathrm{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}.$$

SwiGLU activation function

Transformer ("Attention is all you need")

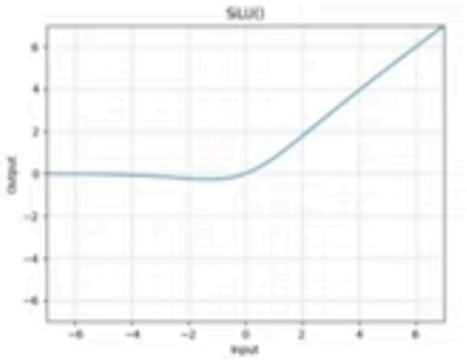
$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

LLaMA

$$\mathrm{FFN}_{\mathrm{SwiGLU}}(x, W, V, W_2) = (\mathrm{Swish}_1(xW) \otimes xV)W_2$$

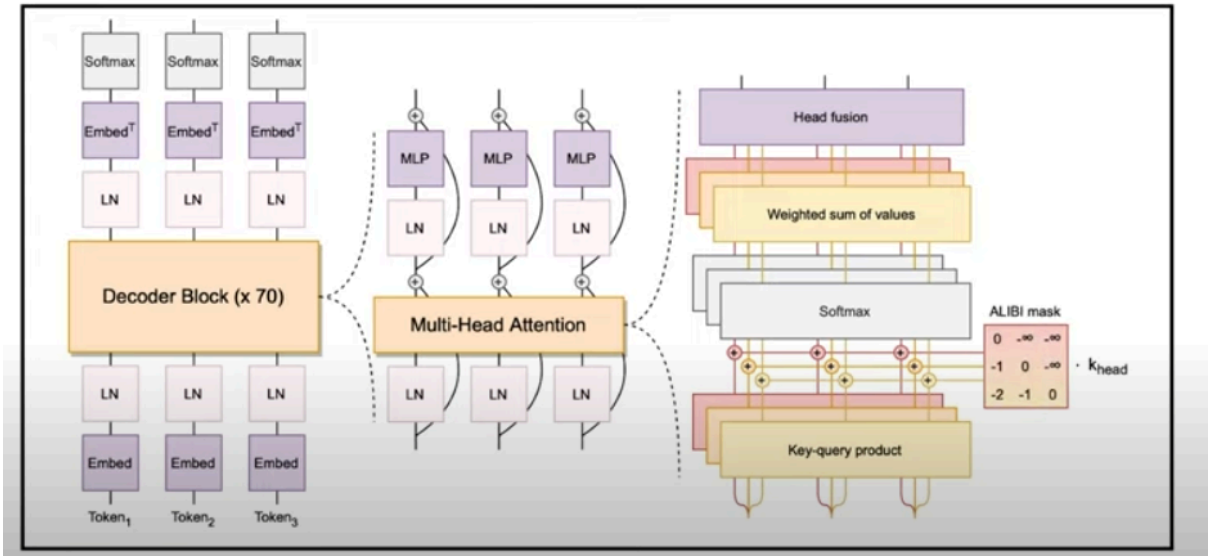
We use the swish function with $\beta = 1$. In this case it's called the **Sigmoid Linear Unit** (**SiLU**) function.

$$\mathrm{swish}(x) = x\,\mathrm{sigmoid}(\beta x) = \frac{x}{1 + e^{-\beta x}}$$



BLOOM

Big science Large Open Science Open Access Multi Language Model.

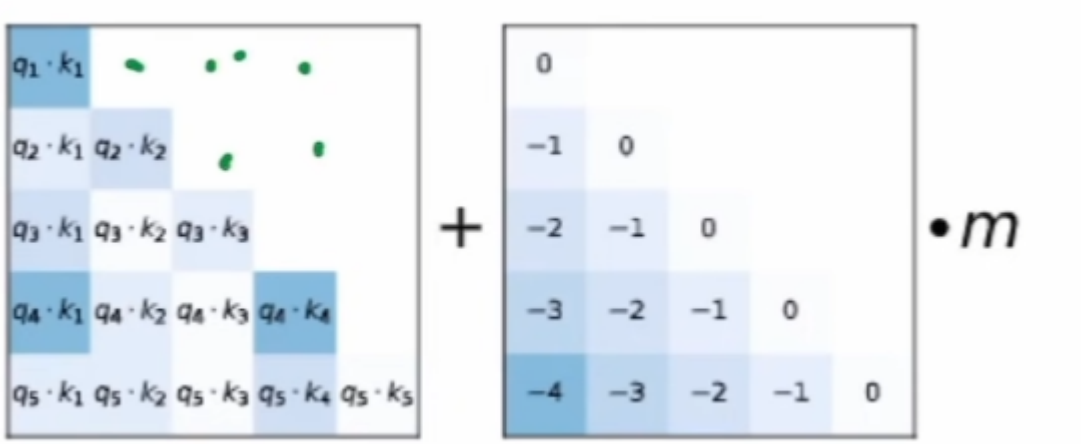


Multitask-prompted finetuning involves training a language model with a mixture of tasks using natural language prompts, as demonstrated by T0 (a part of BigScience), which

showed strong zero-shot task performance after finetuning. T0 used prompts from the Public Pool of Prompts (P3), a collection of 2000+ prompts covering 170+ datasets across various tasks, excluding harmful content. An open-source toolkit called promptsource facilitated prompt creation. BLOOMZ, derived from BLOOM through multitask finetuning, gained multilingual zero-shot capabilities using xP3, an extended dataset covering 46 languages and 16 tasks, with both cross-lingual and monolingual prompts, including machine-translated ones (xP3mt) and added metadata.

ALiBi Positional Embeddings (Attention with Linear Biases)

Instead of adding positional information to the embedding layer, ALiBi directly attenuates the attention scores based on how far away the keys and queries are. Although ALiBi was initially motivated by its ability to extrapolate to longer sequences, it also led to smoother training and better downstream performance even at the original sequence length — outperforming both learned and rotary embeddings.



Embedding LayerNorm

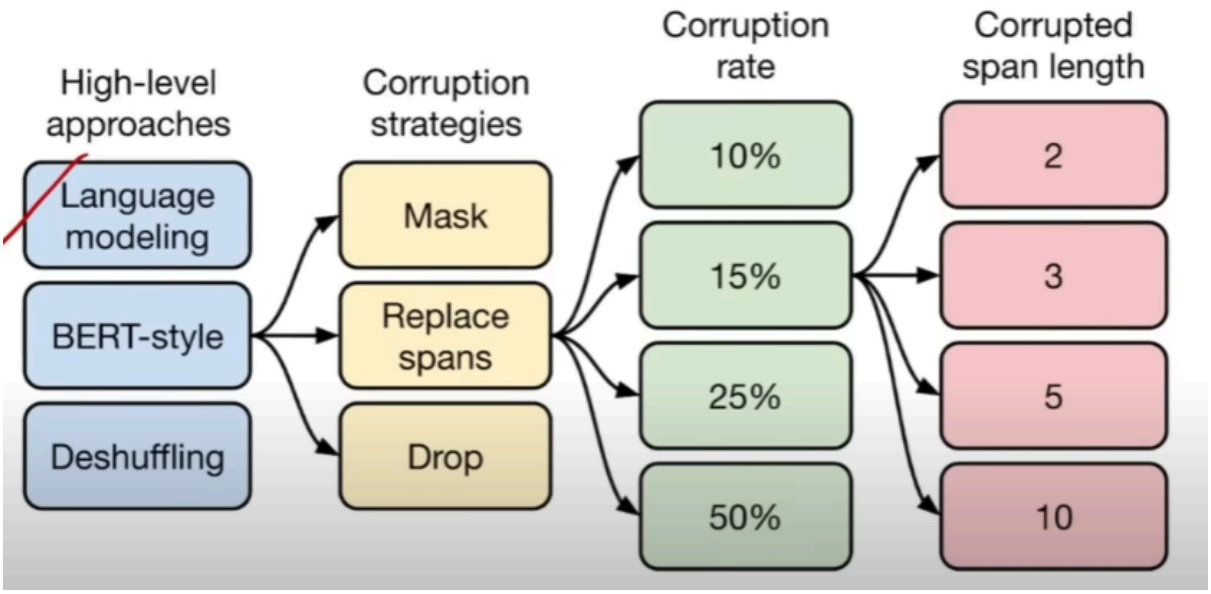
In preliminary experiments training a 104B parameters model, adding an additional layer normalization immediately after the embedding layer significantly improved training stability. Even though it penalizes zero-shot generalization, BLOOM is trained with an embedding layer normalization.

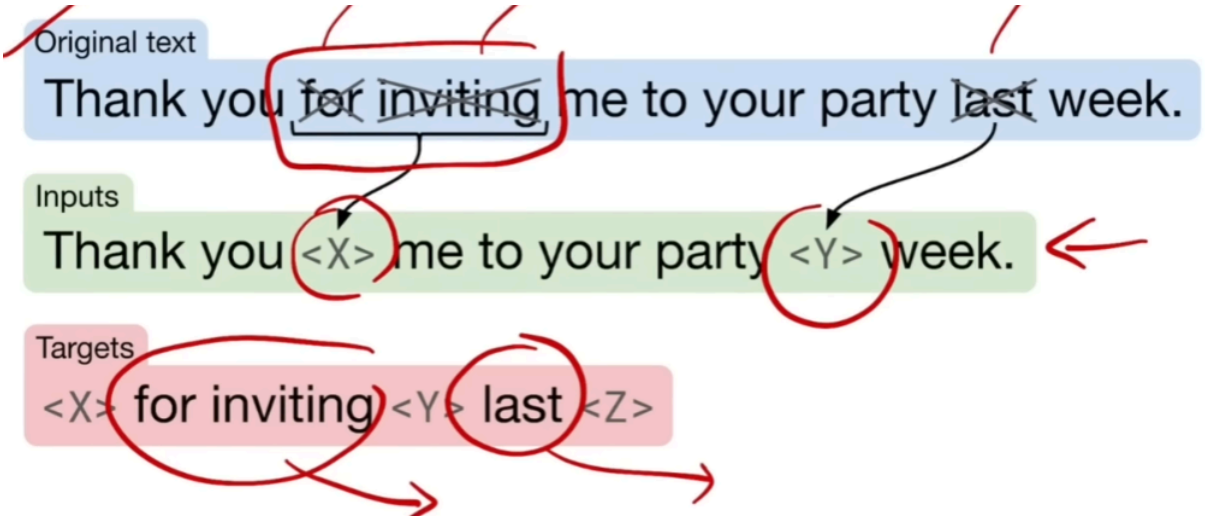
T5

Text to Text Transformer (Encode-Decoder)

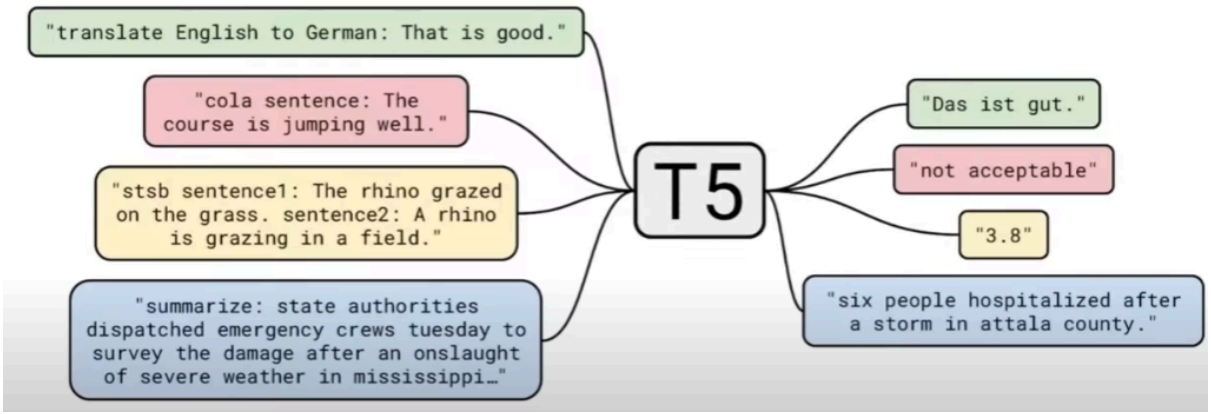
Pretraining

language modeling (predicting the next word), BERT-style objective (which is masking/replacing words with a random different words and predicting the original text), and deshuffling (which is shuffling the input randomly and try to predict the original text)

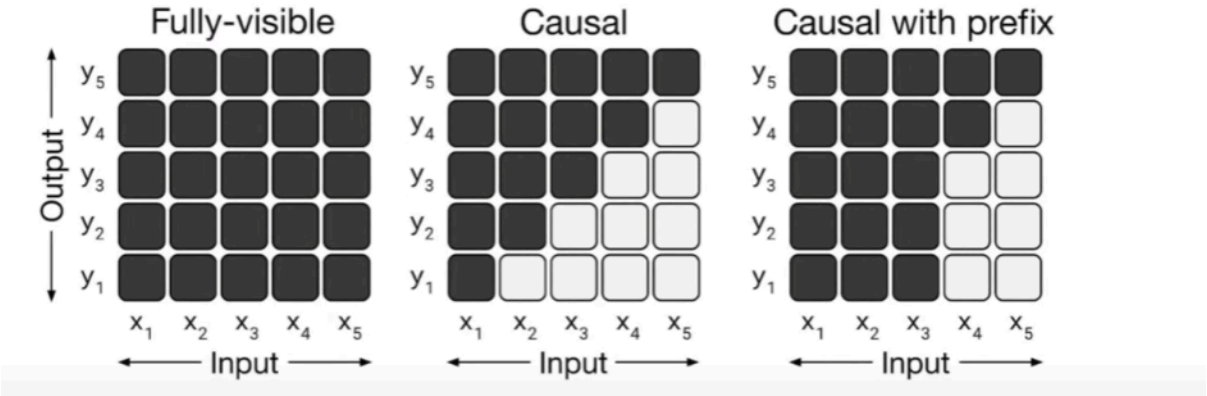




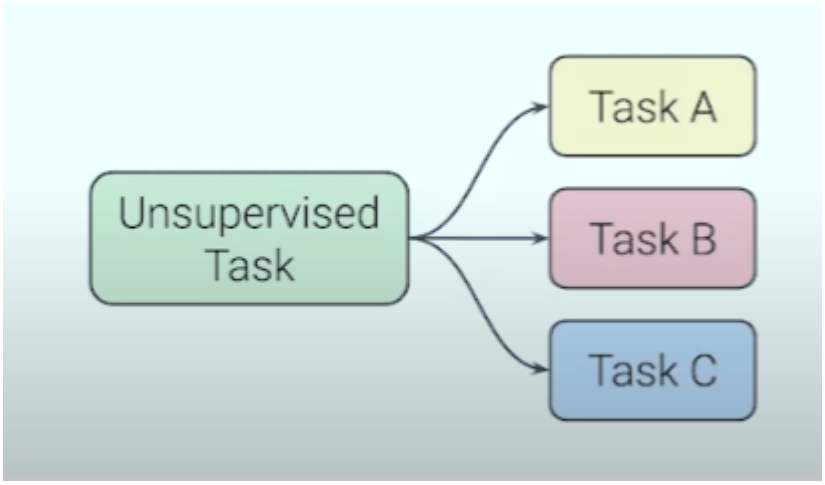
Task specific prefix for various tasks like Translation, Classification, Summarization, QnA e.t.c.,

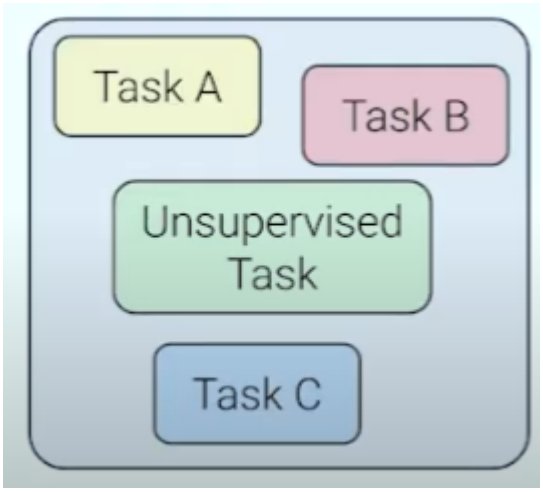


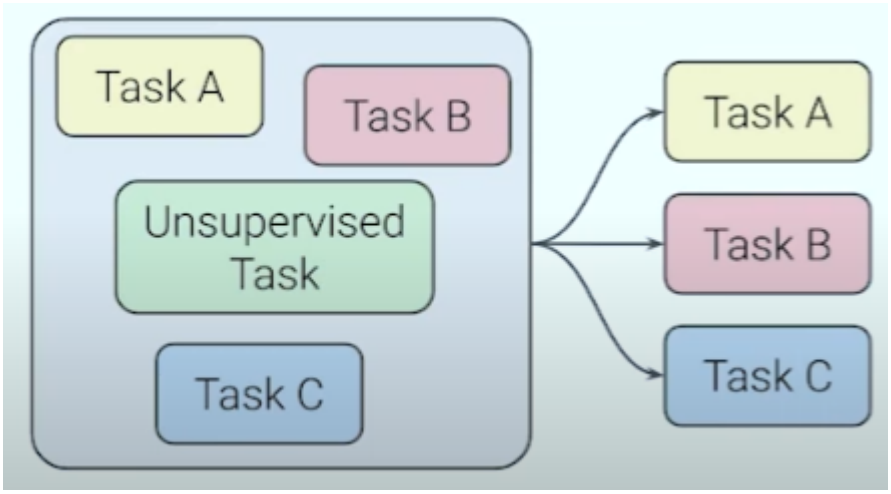
Attention types discussed:

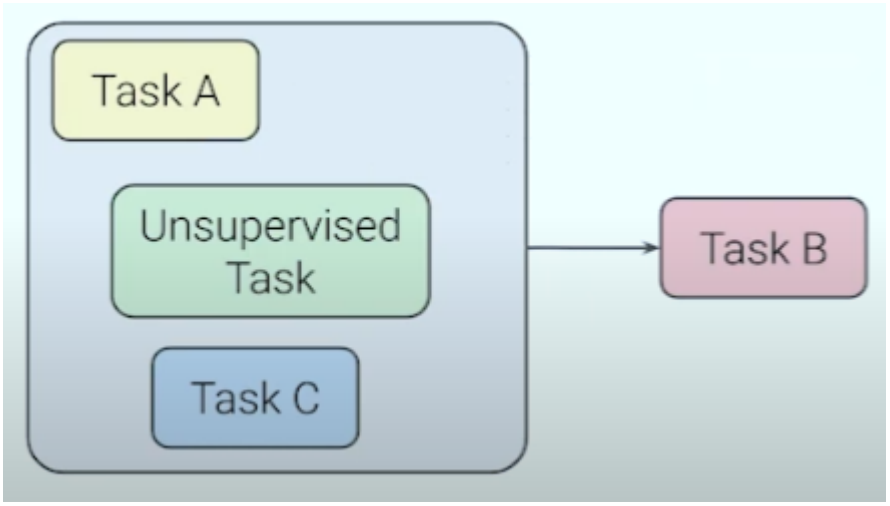


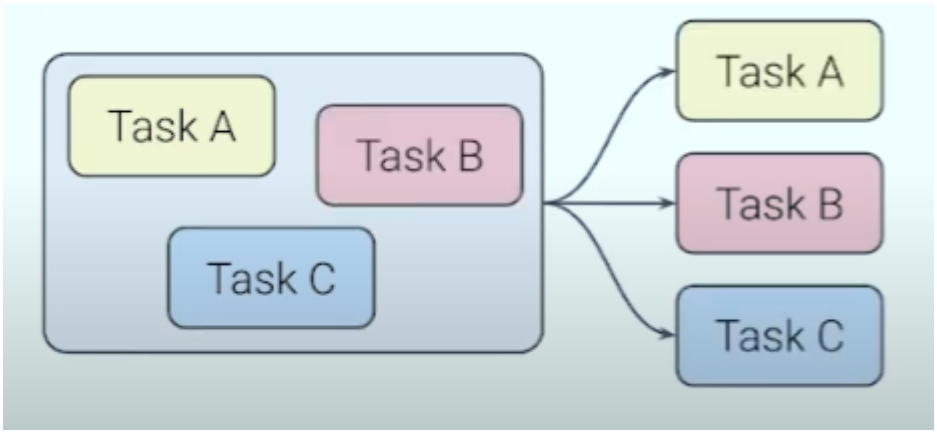
Pretraining and Finetuning strategies







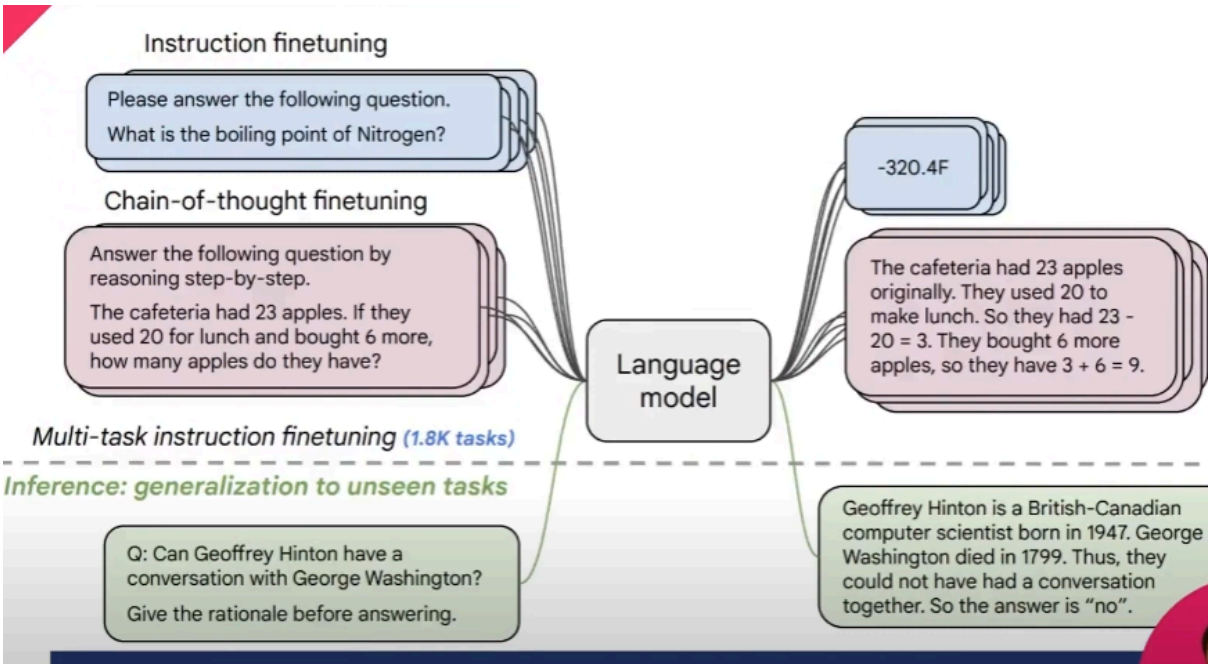




Flan T5:
Flan-T5 is an open-source LLM that's available for commercial usage. Published by Google researchers, Flan-T5 is an encoder-decoder model pre-trained on a variety of language tasks. The model has been trained on supervised and unsupervised datasets with the goal of learning mappings between sequences of text, i.e., text-to-text.

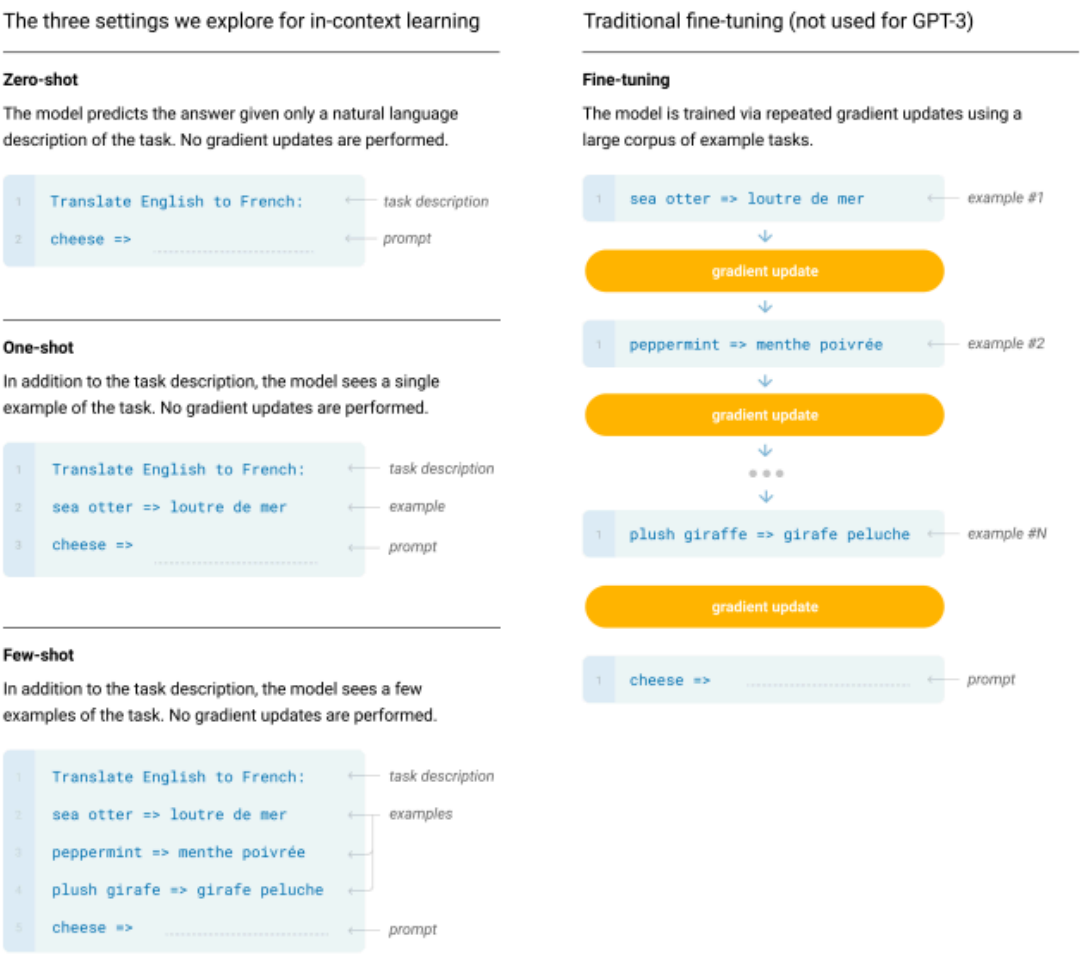
In our view, what sets Flan-T5 apart from other models is that its training is based on prompting. In other words, the model has knowledge of performing specific tasks such as summarization, classification and translation to name a few. For instance, if you were to feed

this blog post into Flan-T5 and tell it to “summarize this article”, it would know that it needs to generate a shorter version of this article. If you download the model from Hugging Face, you can start using it right away for general purpose applications.



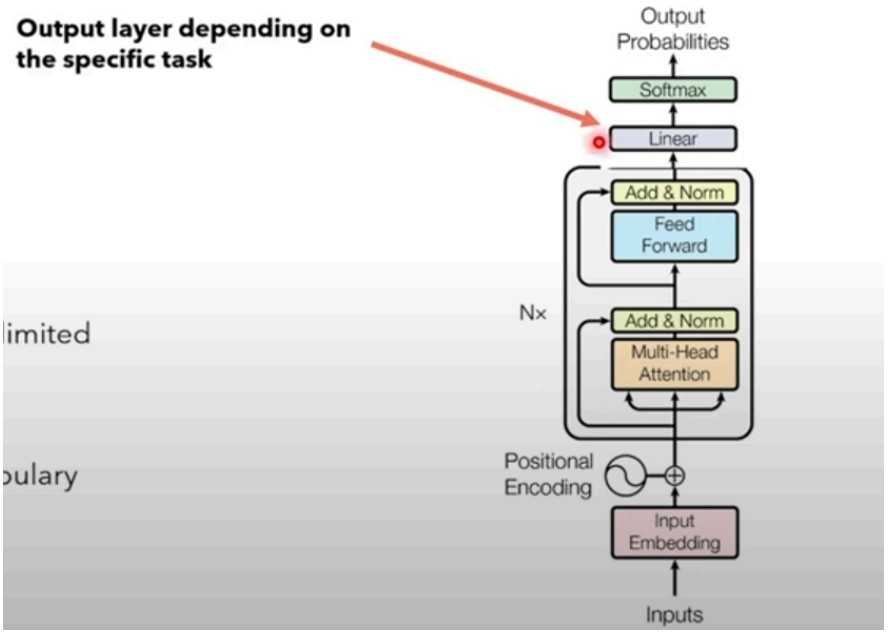
GPT

Language models are few shot learners
Pretraining using Next word prediction task.



BERT

Encoder Only



BERT stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers.

1. Unlike common language models, BERT does not handle "special tasks" with prompts, but rather, it can be specialized on a particular task by means of fine-tuning.
2. Unlike common language models, BERT has been trained using the *left context* and the *right context*.
3. Unlike common language models, BERT is not built specifically for text generation.
4. Unlike common language models, BERT has not been trained on the Next Token Prediction task, but rather, on the Masked Language Model and Next Sentence Prediction task.

Masked Language Model (MLM)

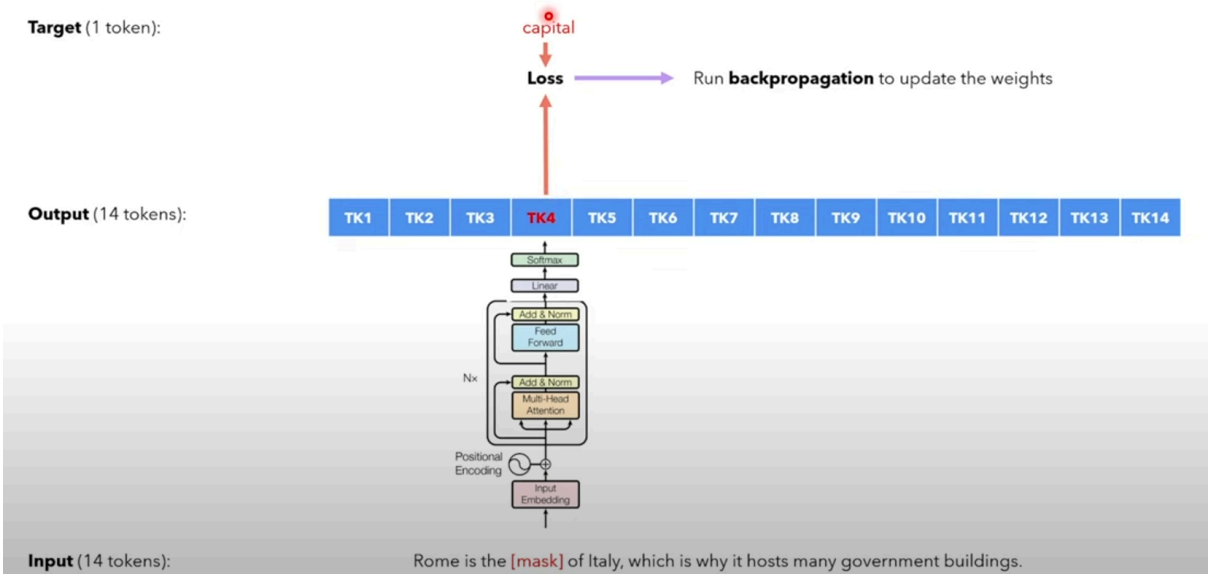
Also known as the Cloze task. It means that randomly selected words in a sentence are masked, and the model must **predict the right word given the left and right context**.

Rome is the capital of Italy, which is why it hosts many government buildings.

Randomly select one or more tokens and replace them with the special token **[MASK]**

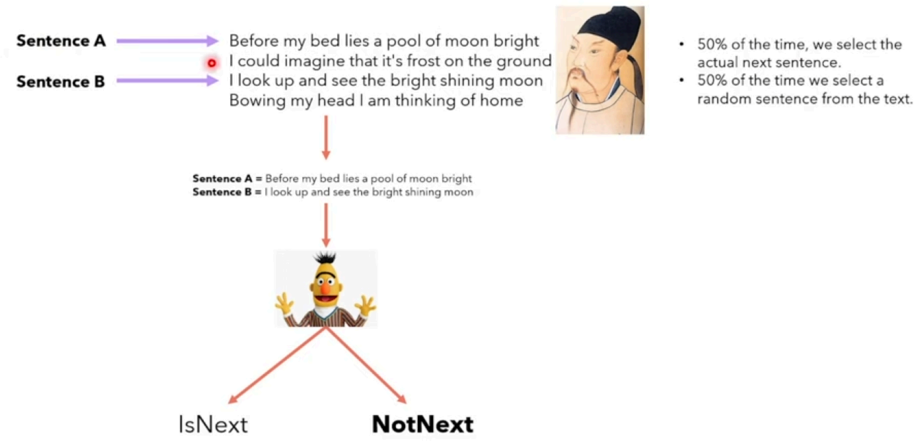
Rome is the [MASK] of Italy, which is why it hosts many government buildings.

capital



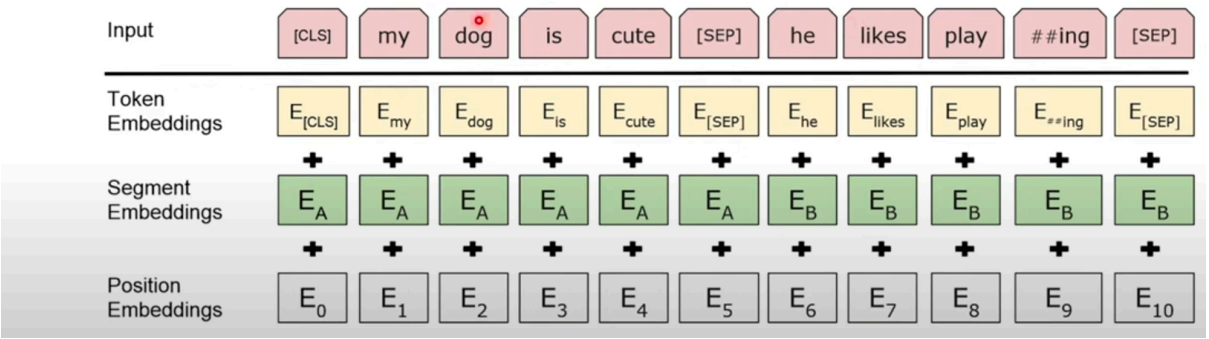
Next Sentence Prediction (NSP)

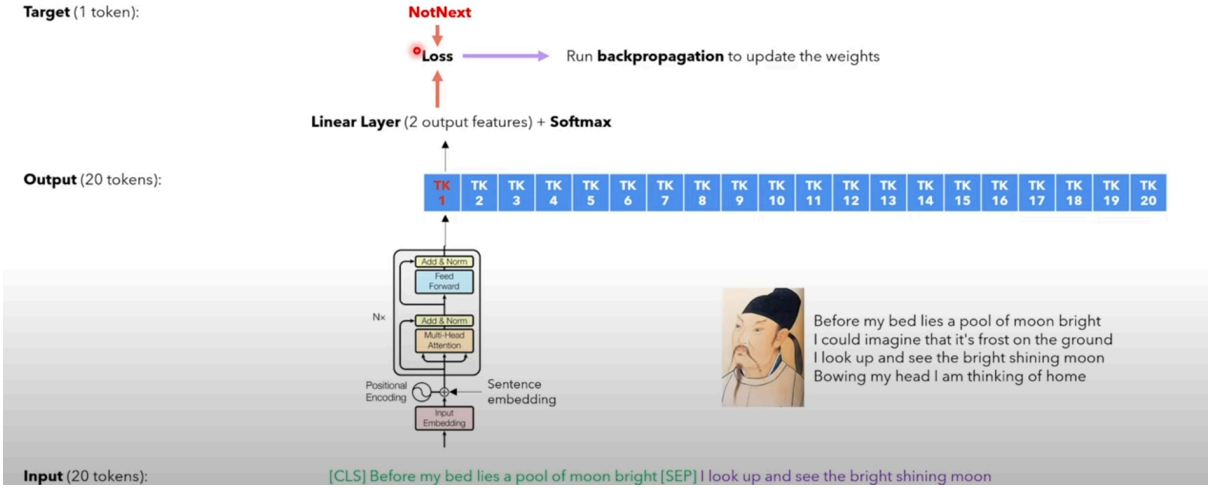
Many downstream applications (for example choosing the right answer given 4 choices) require learning relationships between sentences rather than single tokens, that's why BERT has been pre-trained on the Next Sentence Prediction task.



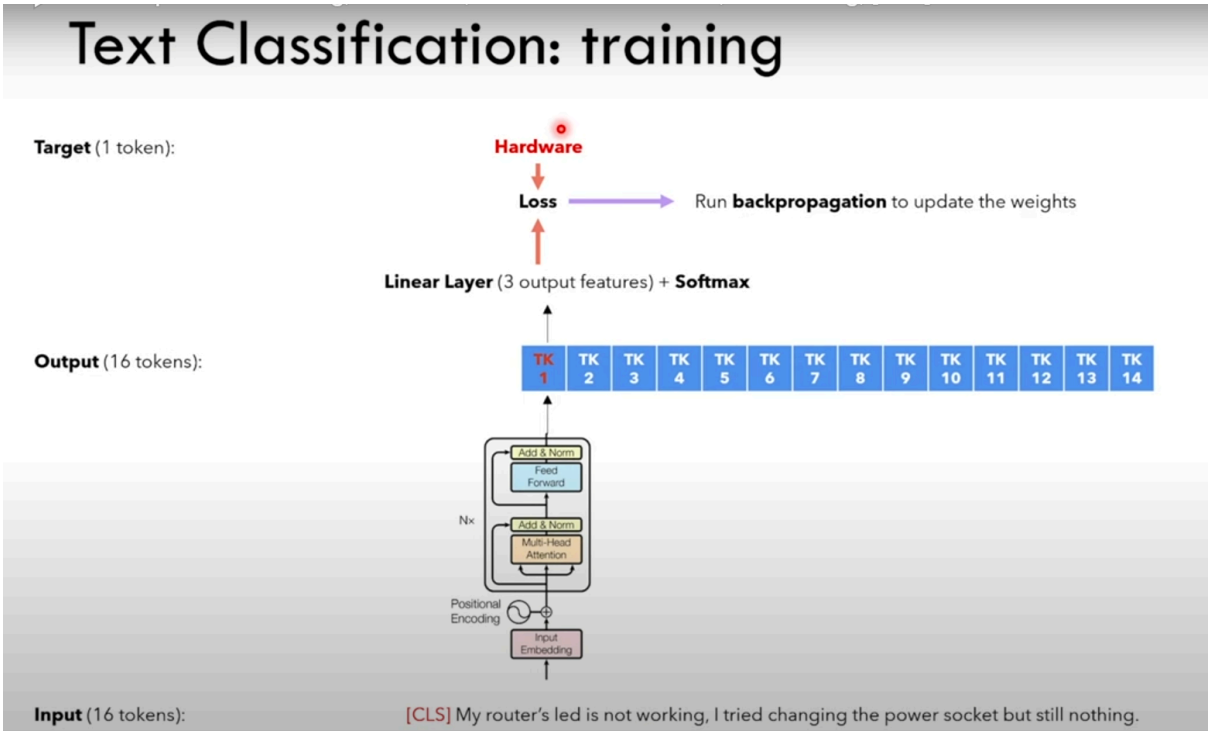
Given the sentence A and the sentence B, how can BERT understand which tokens belongs to the sentence A and which to the sentence B? We introduce the segmentation embeddings!

We also introduce two special tokens: **[CLS]** and **[SEP]**

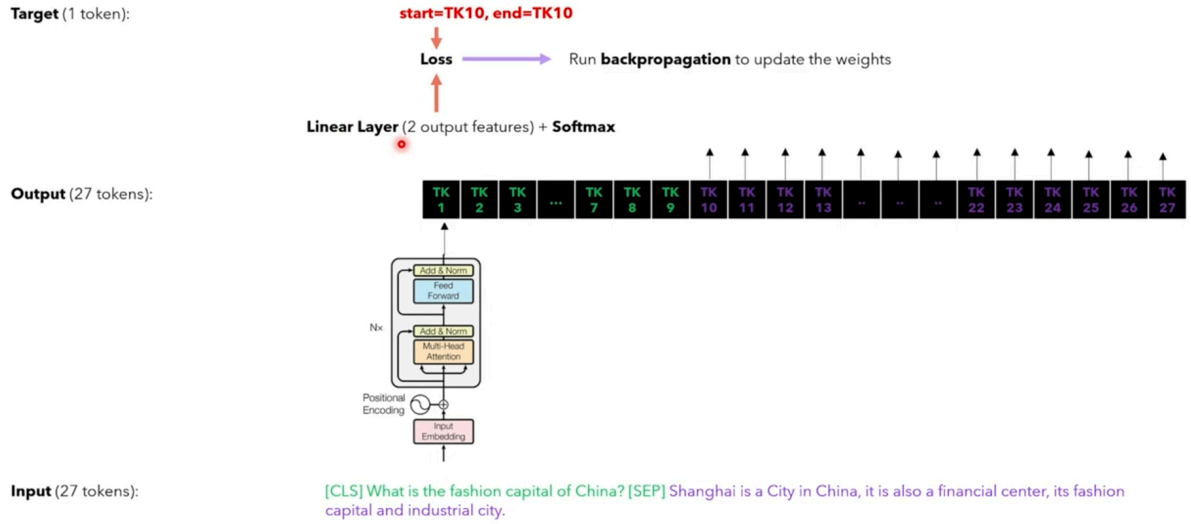




Classification



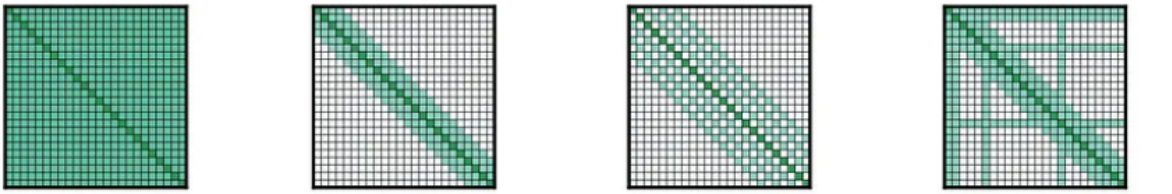
QnA



Longformer

Long-Document Transformer (Encoder only… And one more variant of it is Longformer Encoder-Decoder, LED)

Attends to long documents unlike BERT variants.



(a) Full n^2 attention (b) Sliding window attention (c) Dilated sliding window (d) Global+sliding window

- (a) The original Transformer model: has a self-attention component with O(n²) time and memory complexity where n is the input sequence length.

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

- As we can see, there is a QK^T in the attention operation. This becomes a problem when the model becomes very large, which consumes large amount of memory.
- (b) Sliding Window: This attention pattern employs a fixed-size window attention surrounding each token. Given a fixed window size w, each token attends to (1/2)×w tokens on each side.
- The computation complexity of this pattern is O(n×w), which scales linearly with input sequence length n.
- (c) Dilated Sliding Window: To further increase the receptive field without increasing computation, the sliding window can be “dilated”. Assuming a fixed d and w for all layers, the receptive field is l×d×w, which can reach tens of thousands of tokens even for small values of d.

- Different dilation configurations per head improves performance by allowing some heads without dilation to focus on local context, while others with dilation focus on longer context.
- (d) Global Attention: can be added on few pre-selected input locations. The figure shows an example of a sliding window attention with global attention at a few tokens at custom locations.
- Since the number of such tokens is small, the complexity of the combined local and global attention is still O(n).

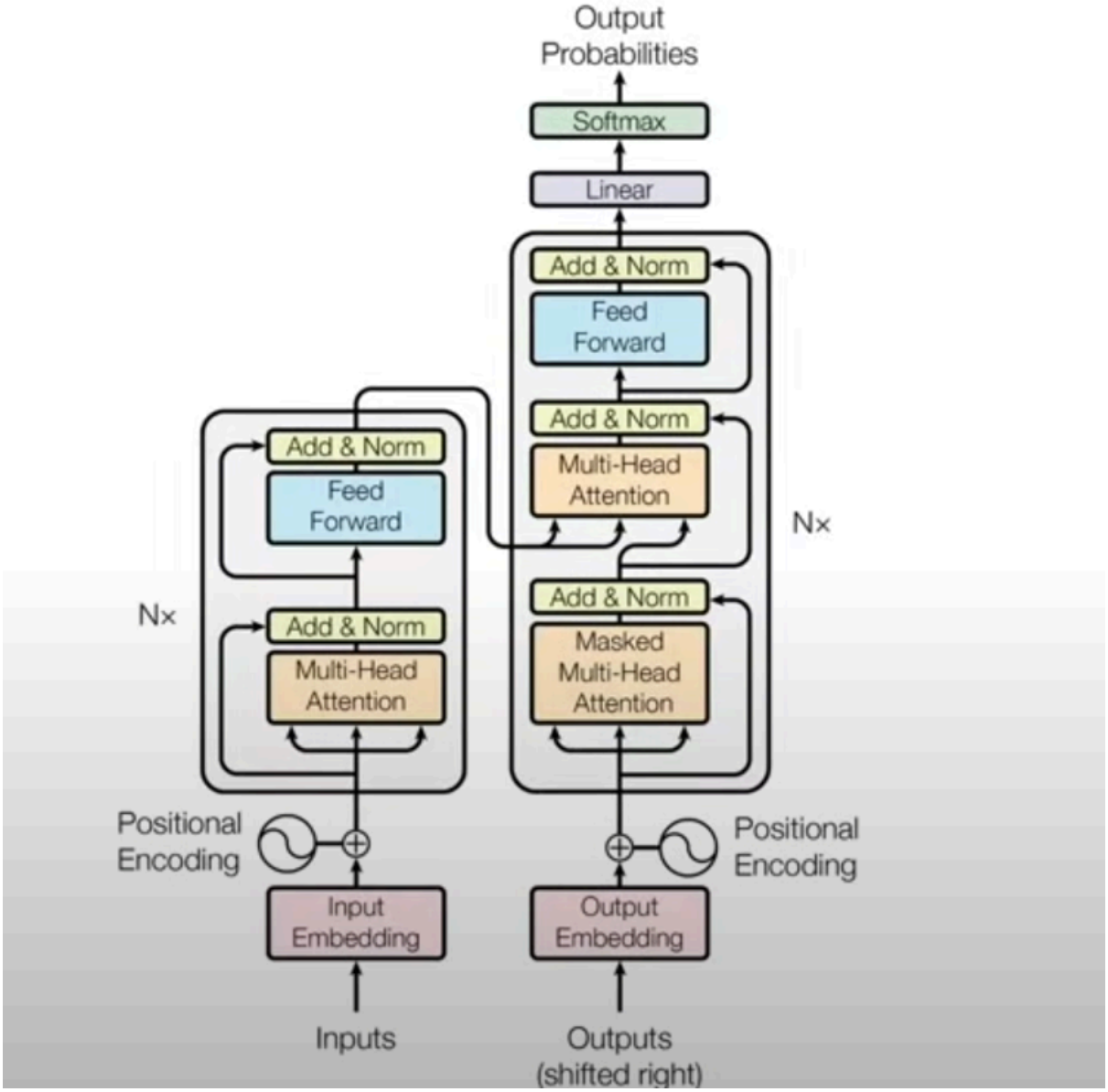
LED

To facilitate modeling long sequences for seq2seq learning, we propose a Longformer variant that has both the encoder and decoder Transformer stacks but instead of the full self-attention in the encoder, it uses the efficient local+global attention pattern of the Longformer. The decoder uses the full self-attention to the entire encoded tokens and to previously decoded locations.

Basic Transformer Architecture Encoder-Decoder

Problems with RNN (among others)

1. Slow computation for long sequences
2. Vanishing or exploding gradients
3. Difficulty in accessing information from long time ago



PEFT, LoRA, QLoRA

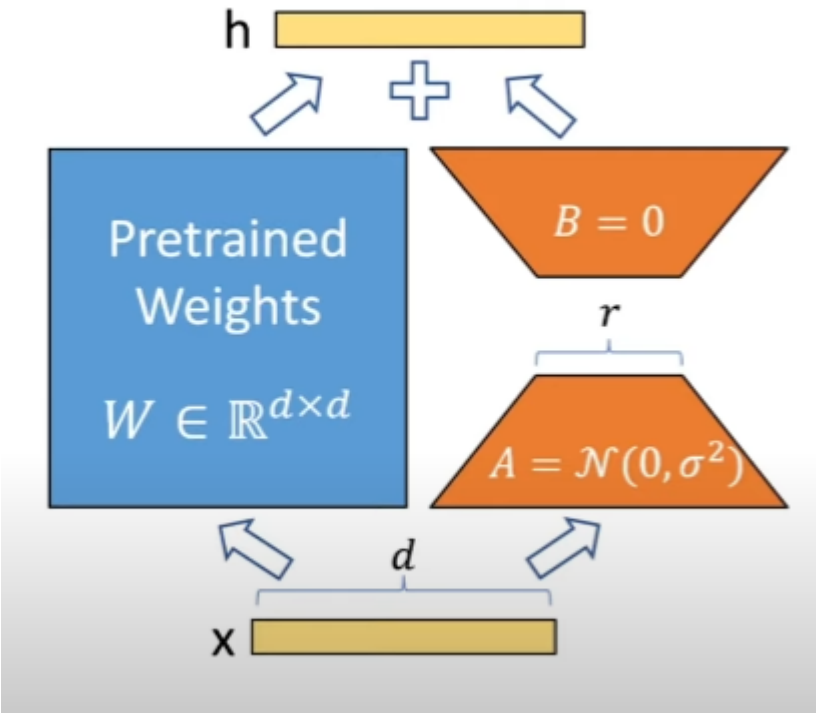
Parameter Efficient Fine Tuning (PEFT)
PEFT is a technique designed to fine-tune models while minimizing the need for extensive resources and cost. PEFT is a great choice when dealing with domain-specific tasks that necessitate model adaptation. By employing PEFT, we can strike a balance between retaining valuable knowledge from the pre-trained model and adapting it effectively to the target task with fewer parameters. There are various ways of achieving Parameter efficient fine-tuning. Low Rank Parameter or LoRA & QLoRA are most widely used and effective.

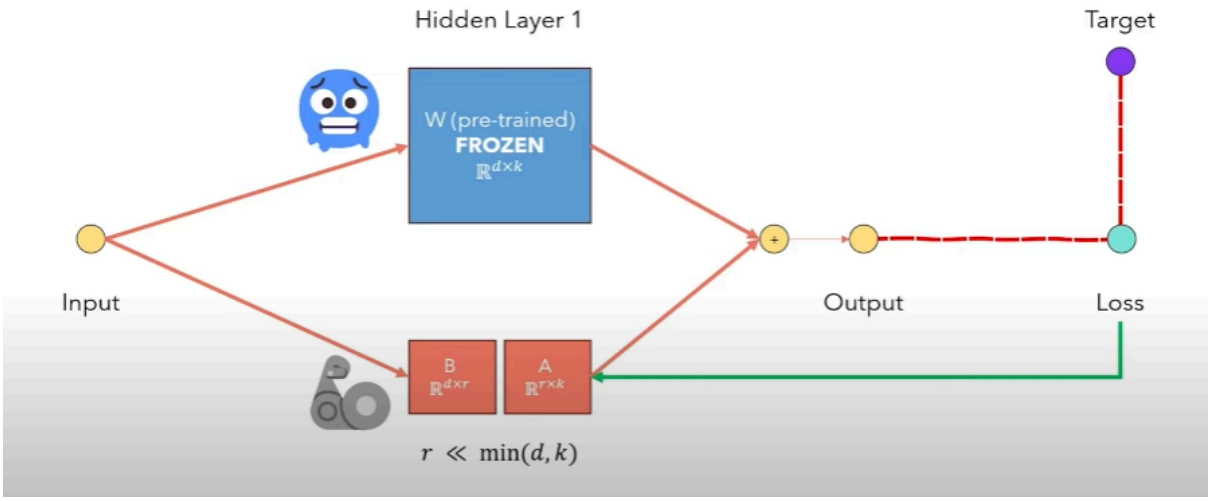
Problems with fine-tuning

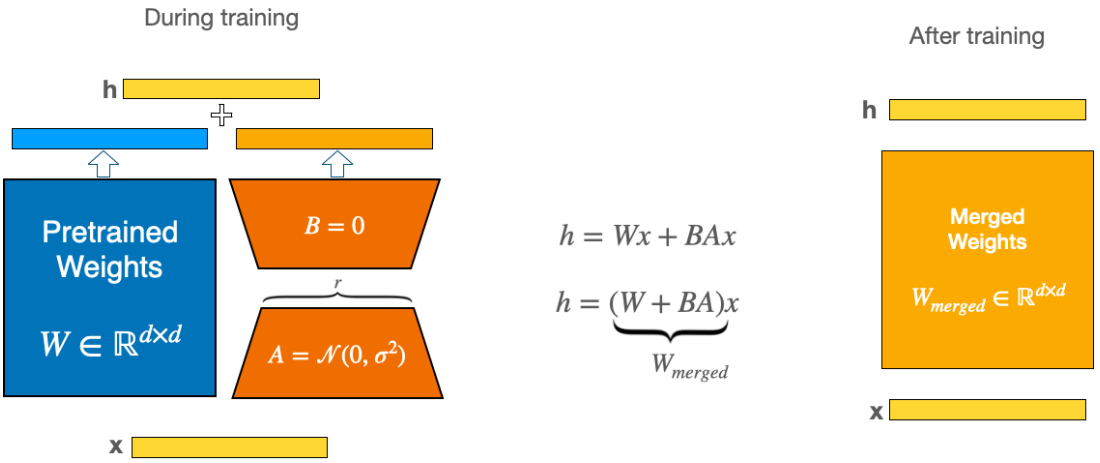
1. We must train the full network, which it computationally expensive for the average user when dealing with Large Language Models like GPT
2. Storage requirements for the checkpoints are expensive, as we need to save the entire model on the disk for each checkpoint. If we also save the optimizer state (which we usually do, then the situation gets worse!)
3. If we have multiple fine-tuned models, we need to reload all the weights of the model every time we want to switch between them, which can be expensive and slow. For example, we may have a model fine-tuned for helping users write SQL queries and one model for helping users write Javascript code.

LORA

Low rank adaptation of Large language models.







QLORA
Quantized LLMs with Low-Rank Adapters

QLoRA is the extended version of LoRA which works by quantizing the precision of the weight parameters in the pre trained LLM to 4-bit precision. Typically, parameters of trained models are stored in a 32-bit format, but QLoRA compresses them to a 4-bit format. This reduces the memory footprint of the LLM, making it possible to finetune it on a single GPU. This method significantly reduces the memory footprint, making it feasible to run LLM models on less powerful hardware, including consumer GPUs.

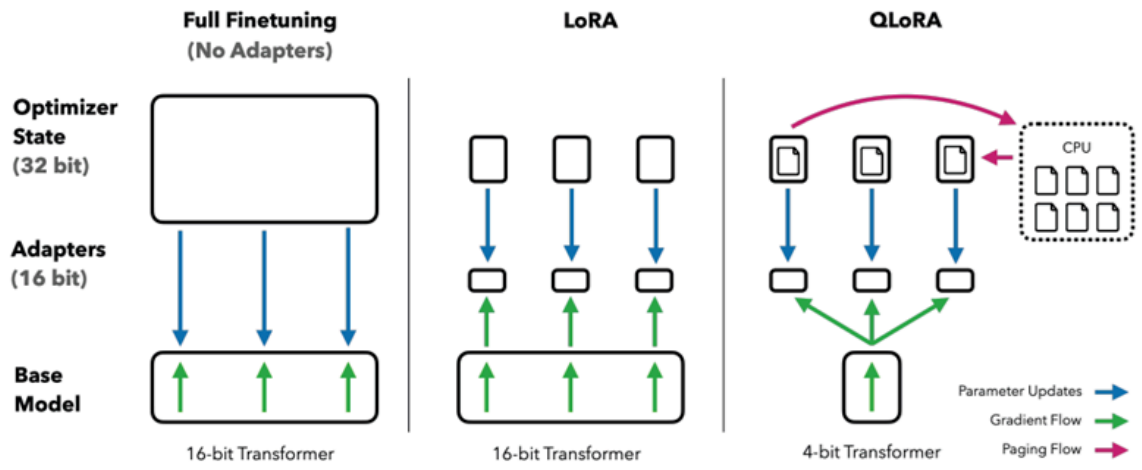


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

According to QLoRA paper:
QLORA introduces multiple innovations designed to reduce memory use without sacrificing performance: (1) 4-bit NormalFloat, an information theoretically optimal quantization data type for normally distributed data that yields better empirical results than 4-bit Integers and 4-bit Floats. (2) Double Quantization, a method that quantizes the quantization constants, saving an average of about 0.37 bits per parameter (approximately 3 GB for a 65B model). (3) Paged Optimizers, using NVIDIA unified memory to avoid the gradient checkpointing memory spikes that occur when processing a mini-batch with a long sequence length.

4-bit Normal Float:
NF4 is a data type specifically designed for AI applications, particularly in the context of quantizing the weights of neural networks to reduce memory footprints of models significantly while attempting to maintain performance. This is crucial for deploying large models on less powerful hardware. NF4 is information-theoretically optimal for data that has a normal distribution, which is a common feature of neural network weights and can represent these weights more accurately than a standard 4-bit float would, within the given bit constraint.
While standard 4-bit Float is more general-purpose floating-point representation and not specifically optimized for specific applications. 4-bit float has very limited precision and range and due to its limitations in precision and range, standard 4-bit floats are less common in AI and machine learning applications, especially for tasks requiring high precision in calculations.
Let's see how the given number is stored in floating point for various floating points datatypes.

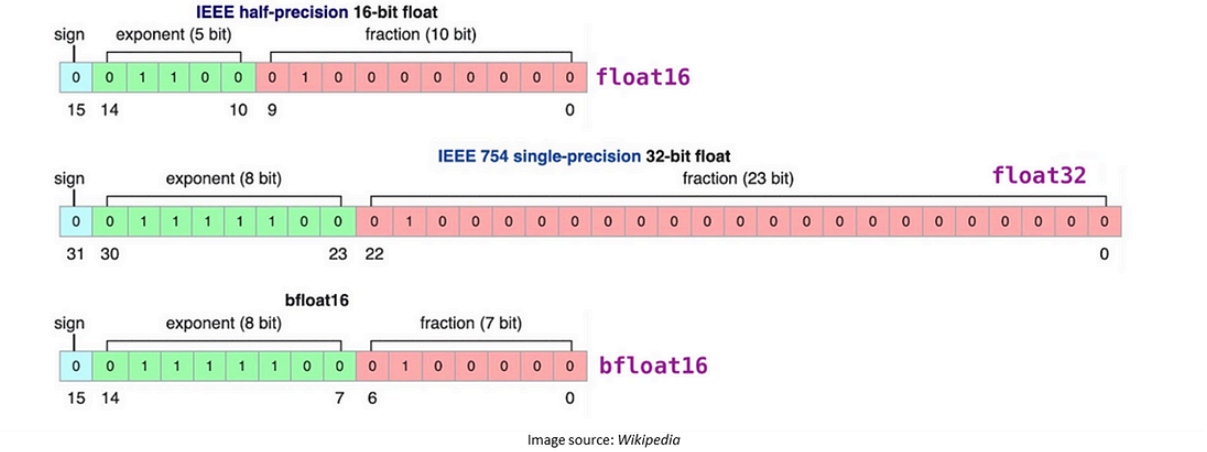


Image source: *Wikipedia*

Each floating point contains 3 different parts which store details about the stored number, that is sign, exponent and fraction also known as mantissa for the given number. The number is first converted into binary format and then stored in the datatype. Each datatype differs in the number of bits they use and hence in their precision and range. For example, FP32 can represent numbers approximately between ±1.18×10^-38 and ±3.4×10^{38}. FP32 is single-precision binary floating point format and has been used as the default format to store weights and biases in Deep Learning. while Fp8 has a range of [-127, 127] and NF4 has a range of [-8, 7].

QLoRA uses brainfloat16 (bfloat16) datatype to perform computational operation that is during forward and backward pass. Brain Floating Point was developed by Google for use in machine learning and other applications that require high throughput of floating-point operations.

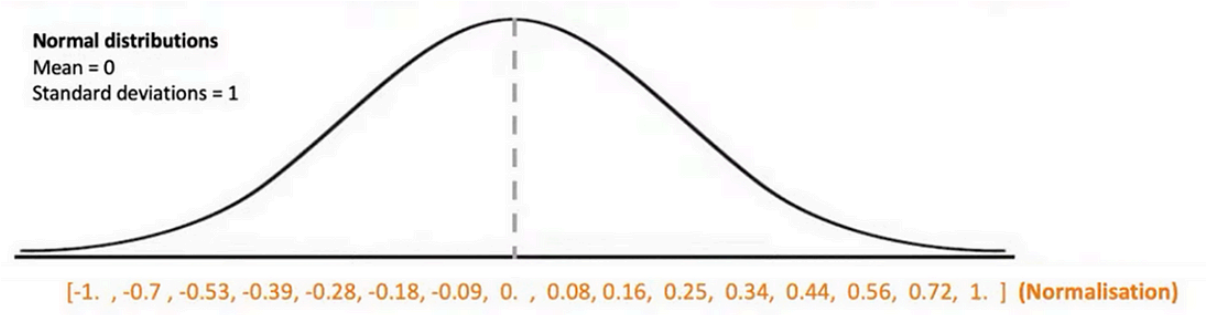
Quantization:

Quantization is a technique that is helpful in reducing the size of the model by converting high precision data to low precision. In simple terms, it converts datatype of high bits to fewer bits. For example, converting FP32 to 8-bit Integers is a quantization technique.

4-bit NormalFloat Quantization:

4-bit NormalFloat Quantization is a method designed to efficiently quantize the weights of neural networks into a 4-bit format. NormalFloat data type is designed to optimally quantize data, particularly for use in neural networks and based on a method called “Quantile Quantization” which ensures that each bin (or category) in the quantization process has an equal number of values from the input data (in this case, the weights of a neural network).Quantiles are essentially cutoff points that divide your data into equal parts based bits of a datatype (for example in NF4 datatype we have 4 bits for quantization, so we have 2^4 = 16 distinct values).

The weights of pretrained neural networks are assumed to follow a zero-centered normal distribution, meaning they are distributed around a central value of zero. The weights are normalized in the range of [-1, 1]. This normalization is achieved by diving each weight by absolute maximum value (this method also called absolute maximum rescaling). By normalizing the input data, we are distributing the weights around zero and the less bits are required to store the exponential data of a tensor/weight parameter.



To quantize a normal distribution into 4-bit datatype we have 16 different values(bins) and to find these values we split the normal distribution into 16 pieces of equal width. Then we take the values in each of these bins and quantize it. The exact values for NF4 data type (16 bins) are as follows:
[-1.0, -0.6961928009986877, -0.5250730514526367, -0.39491748809814453, -0.28444138169288635, -0.18477343022823334, -0.09105003625154495, 0.0, 0.07958029955625534, 0.16093020141124725, 0.24611230194568634, 0.33791524171829224, 0.44070982933044434, 0.5626170039176941, 0.7229568362236023, 1.0]
quantized int4 tensor = round (16/absolute maximum tensor) *F32 tensor

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-1	-0.70	-0.53	-0.39	-0.28	-0.18	-0.09	0	0.08	0.16	0.25	0.34	0.44	0.56	0.72	1

Consider a tensor of 0.686 after normalizing between [-1, 1] and this value is compared to closest bin value of int4 tensor. The closest value is 0.72 and its index is 15. So instead of storing 0.686 Float32 datatype int4 datatype store 15 after quantization.

Problem with outliers:

Outliers are very important in neural networks. Even though model weights are normally distributed, there are some outliers which are very important because removing changing this will affect the quality of the model. However, outlier in the quantization effect the process. For example, in the below figure the outlier at -10 affects the distribution and the bins between -10 and -3 are empty and only 8 quantization bins are filled with values, and this makes quantization equal to 3-bit quantization. This issue might be severe and can degrade the performance.

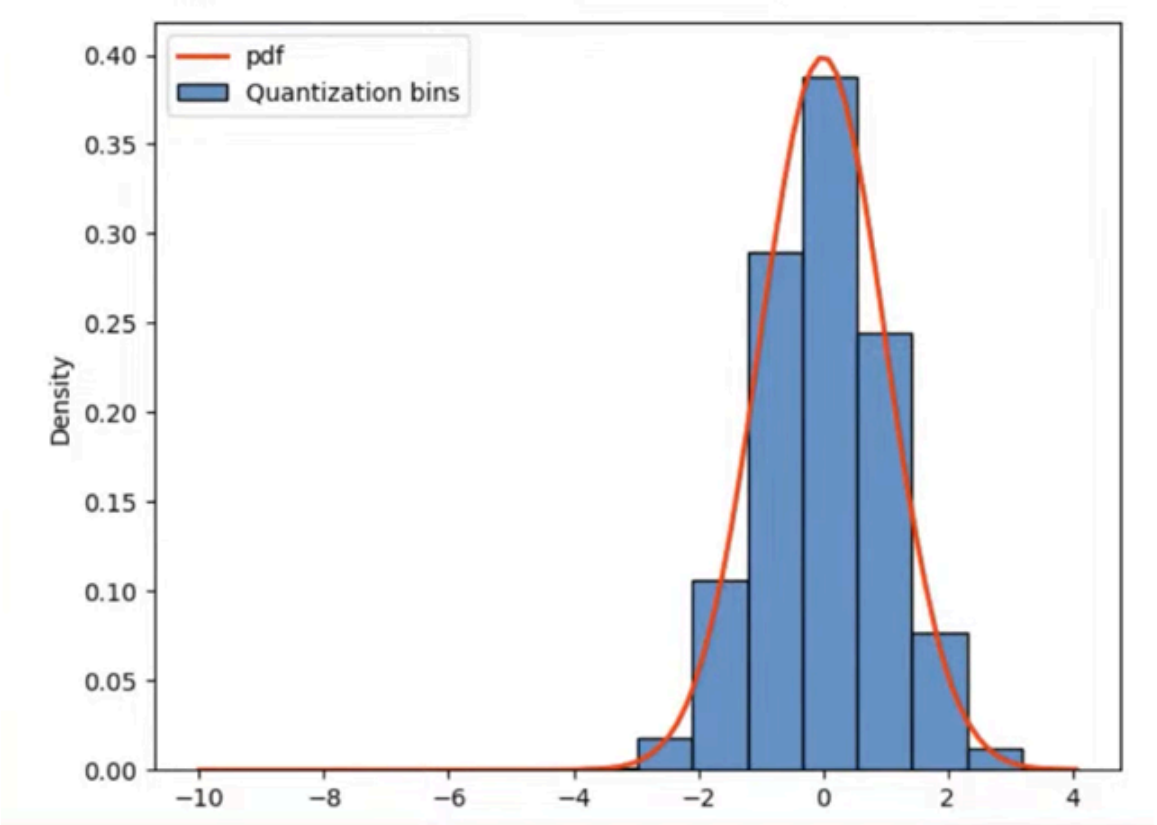


Image source: *Democratizing Foundation Models via k-bit Quantization by Tim Dettmers*

Block wise k-bit quantization:
Block-wise quantization divides input tensors into smaller blocks and quantizes each block independently which reduces the problem of outlier. In this process we split the input tensor into chunks and each chuck is quantized independently with each having their own quantization constant (total bins of datatype/absolute max tensor). Even with outliers we get much higher quantization precision and stability with block wise k-bit quantization by confining outliers to blocks.

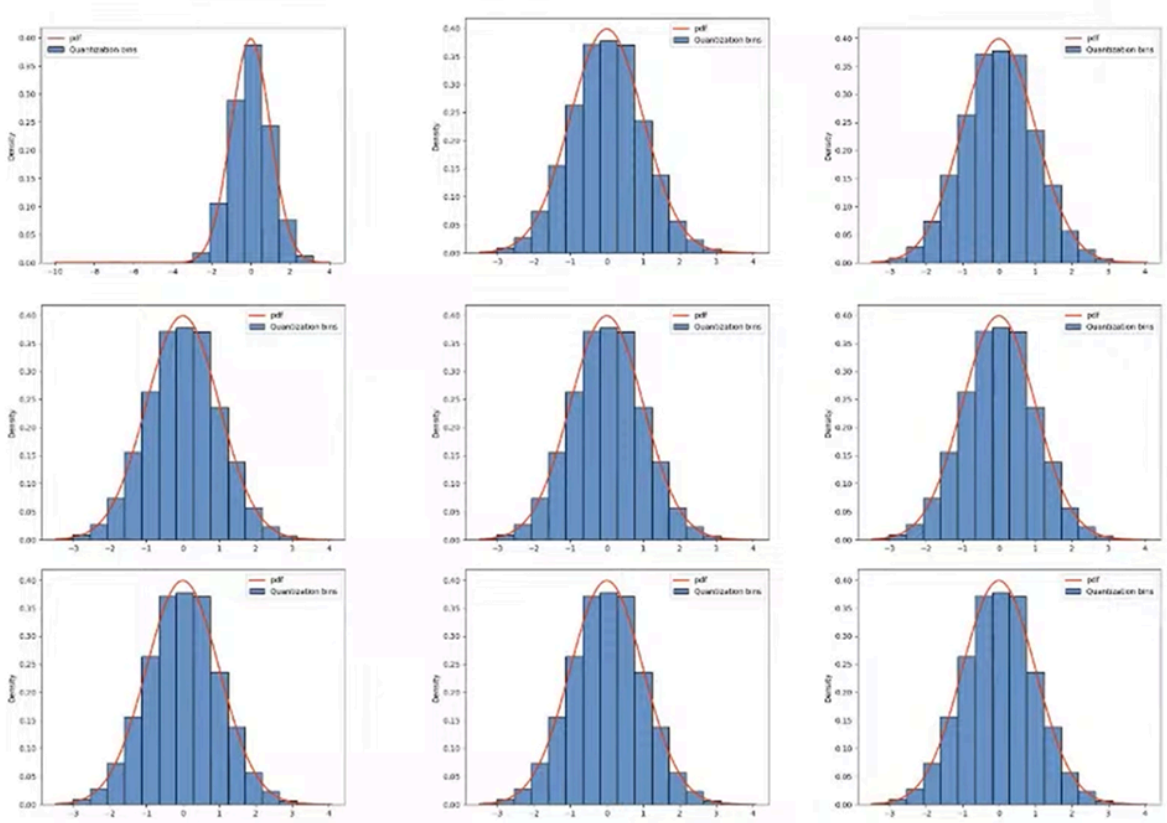


Image source: *Democratizing Foundation Models via k-bit Quantization by Tim Dettmers*

Double quantization:

Double quantization is the process of quantizing the quantization constant to reduce the memory down further to save these constant. To perform dequantization technique we need to store the quantization constants. If we employed blockwise quantization, then we will have n quantization constants in their original datatype. In the case of expansive LLM's which have substantial number of quantization constants that must be stored, leading to increased memory overhead.

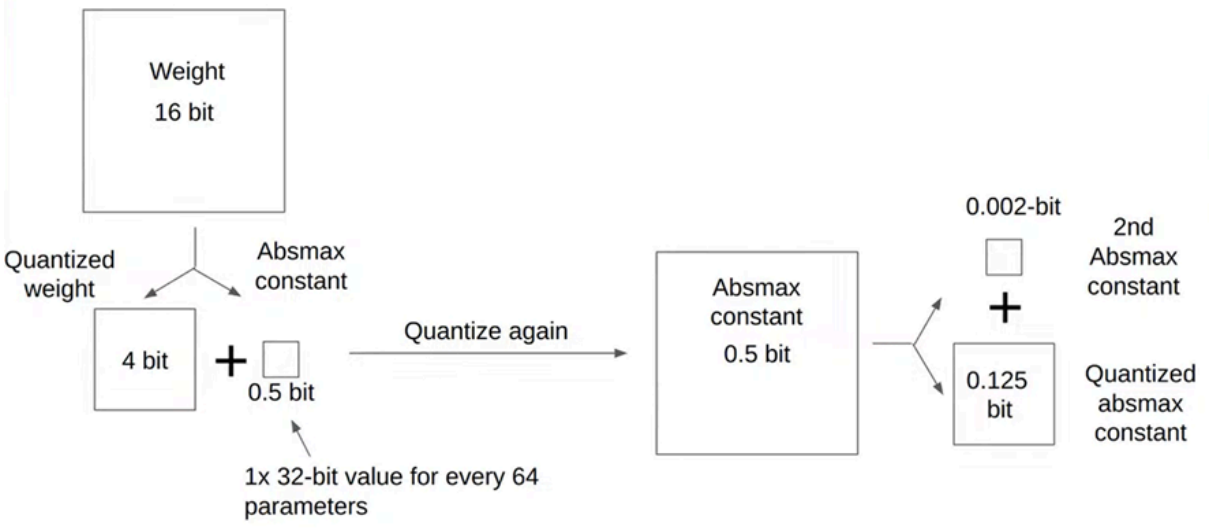


Image source: *Democratizing Foundation Models via k-bit Quantization by Tim Dettmers*

According to QLoRA paper:

For example, using 32-bit constants and a blocksize of 64 for W, quantization constants add 32/64 = 0.5 bits per parameter on average. Double Quantization helps reduce the memory footprint of quantization constants. More specifically, Double Quantization treats quantization constants C2FP32 of the first quantization as inputs to a second quantization. This second step yields the quantized quantization constants C2FP32 and the second level of quantization constants C1FP32. We use 8-bit Floats with a blocksize of 256 for the second quantization as no performance degradation is observed for 8-bit quantization, in line with results from Dettmers and Zettlemoyer [13]. Since the C2FP32 are positive, we subtract the mean from C2 before quantization to center the values around zero and make use of symmetric quantization. On average, for a blocksize of 64, this quantization reduces the memory footprint per parameter from 32/64 = 0.5 bits, to 8/64 + 32/(64 · 256) = 0.127 bits, a reduction of 0.373 bits per parameter.
Even though the memory reduced is only 0.373 per parameter, large models with 70B parameters will have huge impact in reducing the memory footprint.

Dequantization:

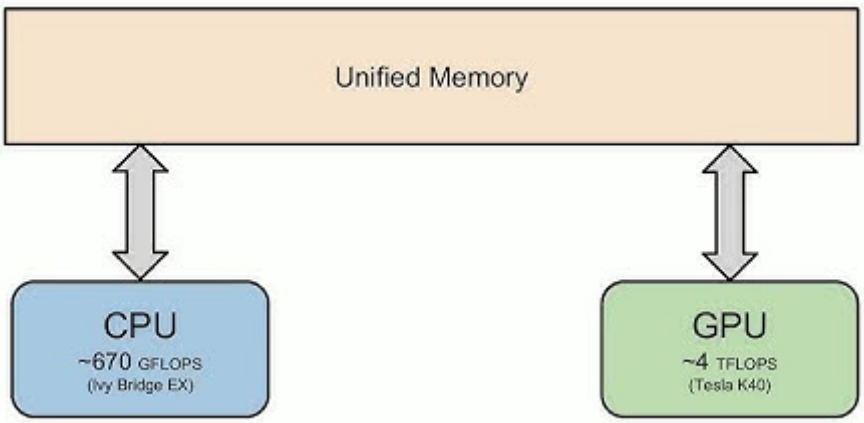
Dequantization is the inverse of quantization. During quantization the weights of the pre-trained model are quantized to smaller data types, such as converting 32-bit data into 4-bit NormalFloat. Quantization significantly reduces the memory requirements for training. However, for inferences and backpropagation during training, these quantized weights (which are frozen and not trained) should be dequantized back to 32-bit for computation. During model fine tuning we need to compute gradients to indicate how each weight should be altered to minimize the loss function. So, in order to perform computation on original datatype we need to implement dequantization technique. If quantized values are used for forward propagation, the calculated gradients become less accurate and unstable. The dequantization is performed using quantization constant and quantized values.

$$\text{dequant}\,(C^{FP32}, X^{NF4}) = X^{NF4}/C^{FP32} = X^{FP32}$$

When we consider the example used in quantization after we dequantize to original datatype we get 0.72 but the original value is 0.749. There is a difference of 0.029, even though this difference looks small but when data moves through several layers this can change the performance of the model. If we incorporated double quantization, then we should also dequantize quantization constant to original tensor(FP32) before using it.

Paged optimizers:

The concept of Paged optimizers is used to manage memory usage during training of large language models (LLMs). When training large models with billions of parameters GPU's running out of memory is common problem. Paged optimizers are used to address these memory spikes that occur during model training.



NVIDIA unified memory facilitates automatic page-to-page transfers between the CPU and GPU, similar to regular memory paging between CPU RAM and the disk. When the GPU runs out of memory, these optimizer states are moved to the CPU RAM and are transferred back into GPU memory when needed.

Small LMs

Large language models (LLMs) have captured headlines and imaginations with their impressive capabilities in natural language processing. However, their massive size and resource requirements have limited their accessibility and applicability. Enter the small language model (SLM), a compact and efficient alternative poised to democratize AI for diverse needs.

What are Small Language Models?

SLMs are essentially smaller versions of their LLM counterparts. They have significantly fewer parameters, typically ranging from a few million to a few billion, compared to LLMs with hundreds of billions or even trillions. This difference in size translates to several advantages:

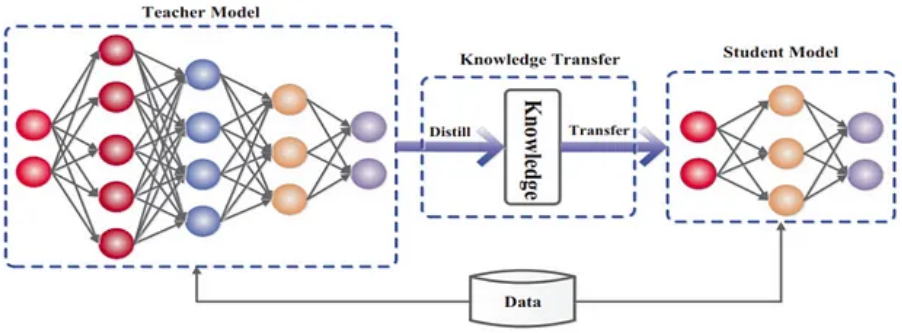
- Efficiency: SLMs require less computational power and memory, making them suitable for deployment on smaller devices or even edge computing scenarios. This opens up opportunities for real-world applications like on-device chatbots and personalized mobile assistants.
- Accessibility: With lower resource requirements, SLMs are more accessible to a broader range of developers and organizations. This democratizes AI, allowing smaller teams and individual researchers to explore the power of language models without significant infrastructure investments.
- Customization: SLMs are easier to fine-tune for specific domains and tasks. This enables the creation of specialized models tailored to niche applications, leading to higher performance and accuracy.

How do Small Language Models Work?

Like LLMs, SLMs are trained on massive datasets of text and code. However, several techniques are employed to achieve their smaller size and efficiency:

Knowledge Distillation:

This involves transferring knowledge from a pre-trained LLM to a smaller model, capturing its core capabilities without the full complexity.

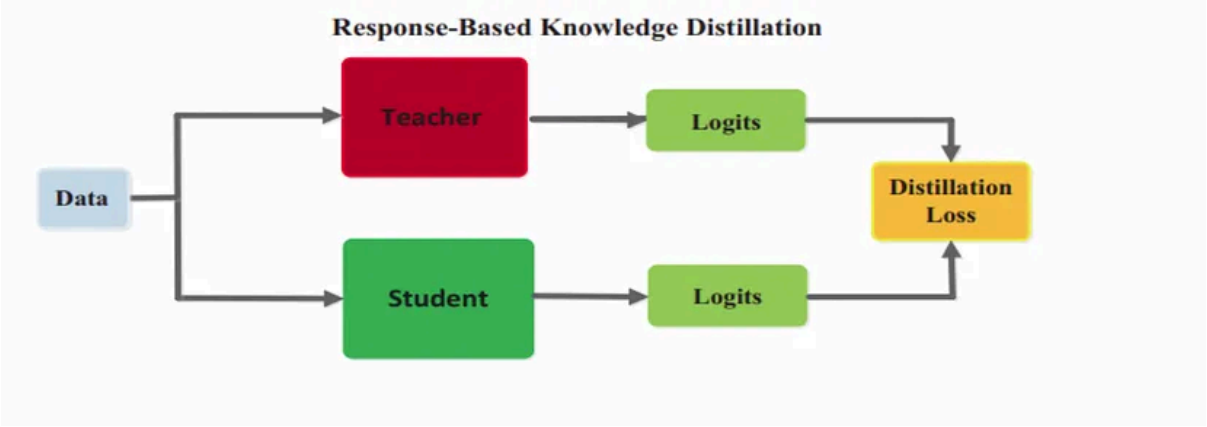


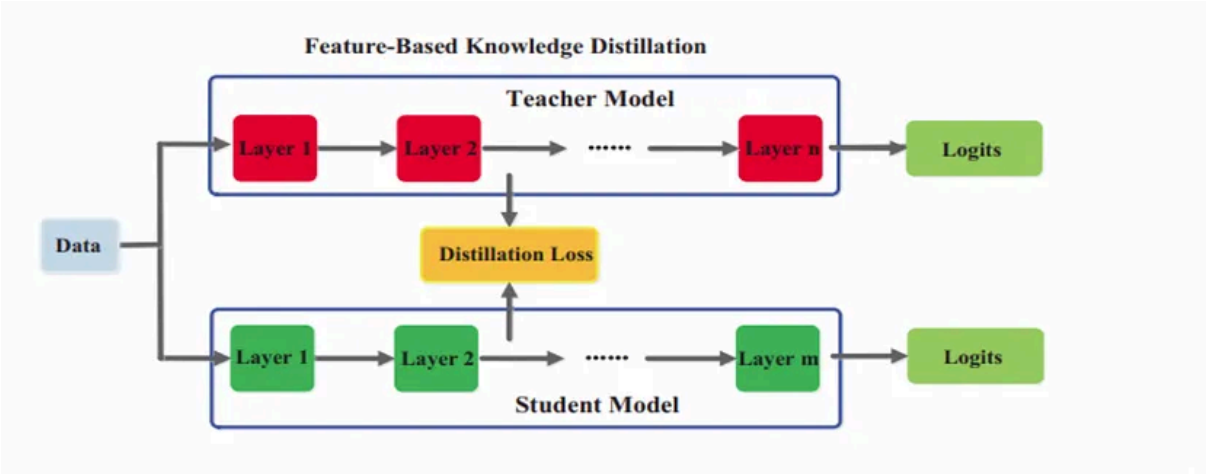
Knowledge distillation refers to the process of transferring knowledge from a large model to a smaller one. This is vital because the larger knowledge capacity of bigger models may not be utilized to its full potential on its own. Even if a model only employs a small percentage of its knowledge capacity, evaluating it can be computationally expensive. Knowledge distillation is the process of moving knowledge from a large model to a smaller one while maintaining validity.
As illustrated in the figure above, knowledge distillation involves a small “student” model learning to mimic a large “teacher” model and using the teacher’s knowledge to achieve similar or superior accuracy.

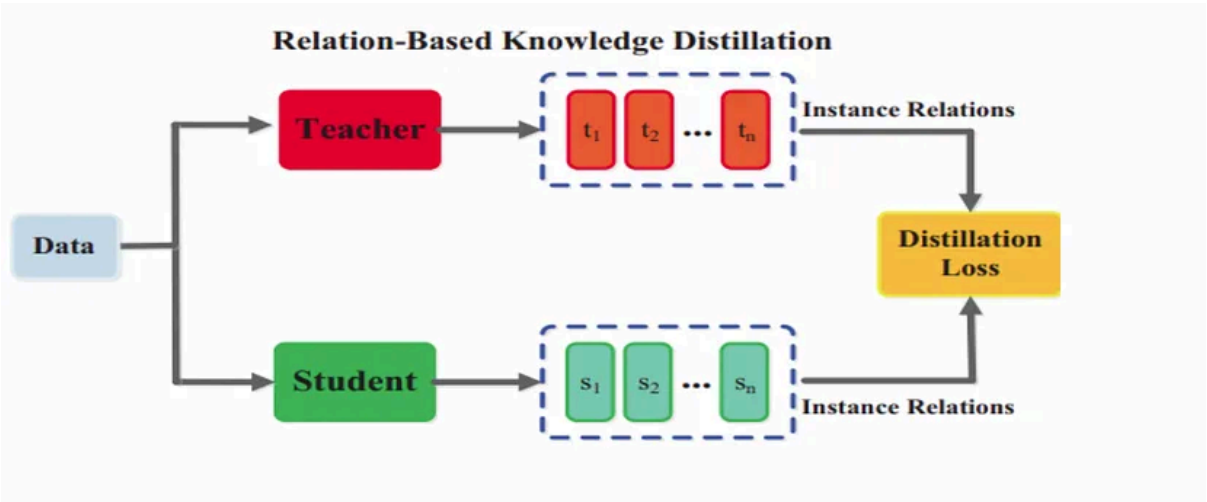
Need for Knowledge Distillation
The size of neural networks is enormous (millions/billions of parameters), necessitating the use of computers with significant memory and computation capability to train/deploy them. In most cases, models must be implemented on systems with little computing power, such as mobile devices and edge devices, in various applications.
However, ultra-light (a few thousand parameters) models may not provide us with good accuracy. This is where Knowledge Distillation comes into play — with assistance from the instructor network, it essentially lightens the model while keeping accuracy.

During training, the loss function will not only consider the difference between the output of the distilled model and the ground truth labels but also the difference between the output of the distilled model and the output of the CNN for the same input.

Types of Knowledge Distillation







Pruning and Quantization:
These techniques remove unnecessary parts of the model and reduce the precision of its weights, respectively, further reducing its size and resource requirements.
Efficient Architectures:
Researchers are continually developing novel architectures specifically designed for SLMs, focusing on optimizing both performance and efficiency.

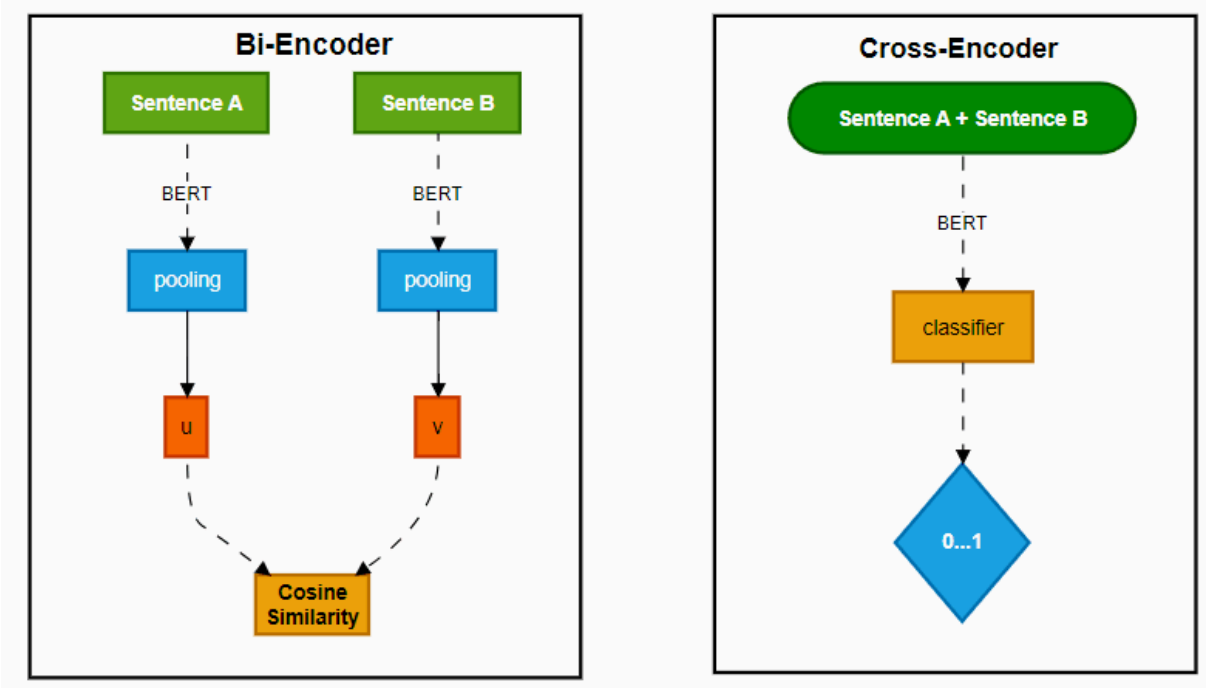
Benefits and Limitations
Small Language Models (SLMs) offer the advantage of being trainable with relatively modest datasets. Their simplified architectures enhance interpretability, and their compact size facilitates deployment on mobile devices.
A notable benefit of SLMs is their capability to process data locally, making them particularly valuable for Internet of Things (IoT) edge devices and enterprises bound by stringent privacy and security regulations.
However, deploying small language models involves a trade-off. Due to their training on smaller datasets, SLMs possess more constrained knowledge bases compared to their Large Language Model (LLM) counterparts. Additionally, their understanding of language

and context tends to be more limited, potentially resulting in less accurate and nuanced responses when compared to larger models.

Criteria	Small Language Models (SLMs)	Large Language Models (LLMs)
Number of Parameters	Few million to a few hundred million	Billions of parameters
Computational Requirements	Lower, suitable for resource-constrained devices	Higher, require substantial computational resources
Ease of Deployment	Easier to deploy on resource-constrained devices	Challenging to deploy due to high resource requirements
Training and Inference Speed	Faster, more efficient	Slower, more computationally intensive
Performance	Competitive, but may not match state-of-the-art results on certain tasks	State-of-the-art performance on various NLP tasks
Model Size	Significantly smaller, typically 40% to 60% smaller than LLMs	Large, requiring substantial storage space
Real-world Applications	Suitable for applications with limited computational resources	Primarily used in resource-rich environments, such as cloud services and high-performance computing systems

Sentence Transformers

The transformer models had one issue when building sentence vectors: Transformers work using word or token-level embeddings, not sentence-level embeddings.
Before sentence transformers, the approach to calculating accurate sentence similarity with BERT was to use a cross-encoder structure. This meant that we would pass two sentences to BERT, add a classification head to the top of BERT — and use this to output a similarity score.
The BERT cross-encoder architecture consists of a BERT model which consume sentences A and B. Both are processed in the same sequence, separated by a [SEP] token. All of this is followed by a feedforward NN classifier that outputs a similarity score.



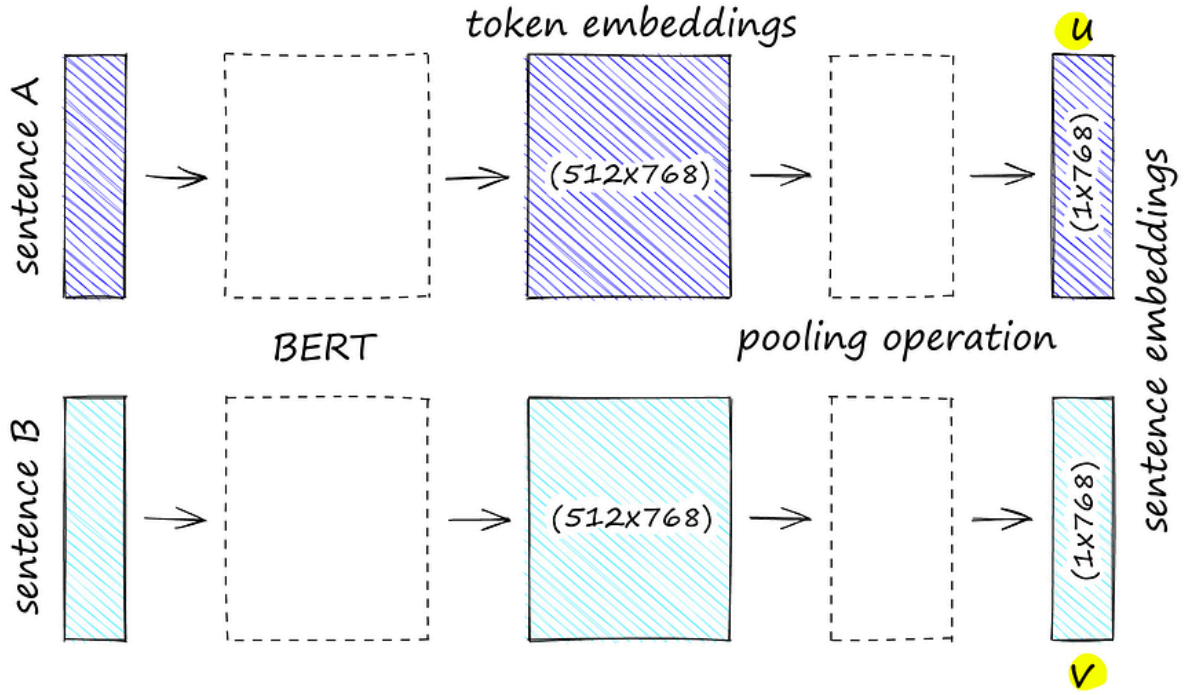
The cross-encoder network does produce very accurate similarity scores (better than SBERT), but it's not scalable. If we wanted to perform a similarity search through a small 100K sentence dataset, we would need to complete the cross-encoder inference computation 100K times.
To cluster sentences, we would need to compare all sentences in our 100K dataset, resulting in just under 500M comparisons — this is simply not realistic.

Ideally, we need to pre-compute sentence vectors that can be stored and then used whenever required. If these vector representations are good, all we need to do is calculate the cosine similarity between each. With the original BERT (and other transformers), we can build a sentence embedding by averaging the values across all token embeddings output by BERT (if we input 512 tokens, we output 512 embeddings). [Approach — 1]
Alternatively, we can use the output of the first [CLS] token (a BERT-specific token whose output embedding is used in classification tasks). [Approach — 2]
Using one of these two approaches gives us our sentence embeddings that can be stored and compared much faster, shifting search times from 65 hours to around 5 seconds. However, the accuracy is not good, and is worse than using averaged GloVe embeddings (which were developed in 2014)

Thus, finding the most similar sentence pair from 10K sentences took 65 hours with BERT. With SBERT, embeddings are created in ~5 seconds and compared with cosine similarity in ~0.01 seconds.
Since the SBERT paper, many more sentence transformer models have been built using similar concepts that went into training the original SBERT. They're all trained on many similar and dissimilar sentence pairs.
Using a loss function such as softmax loss, multiple negatives ranking loss, or MSE margin loss these models are optimized to produce similar embeddings for similar sentences, and dissimilar embeddings otherwise.

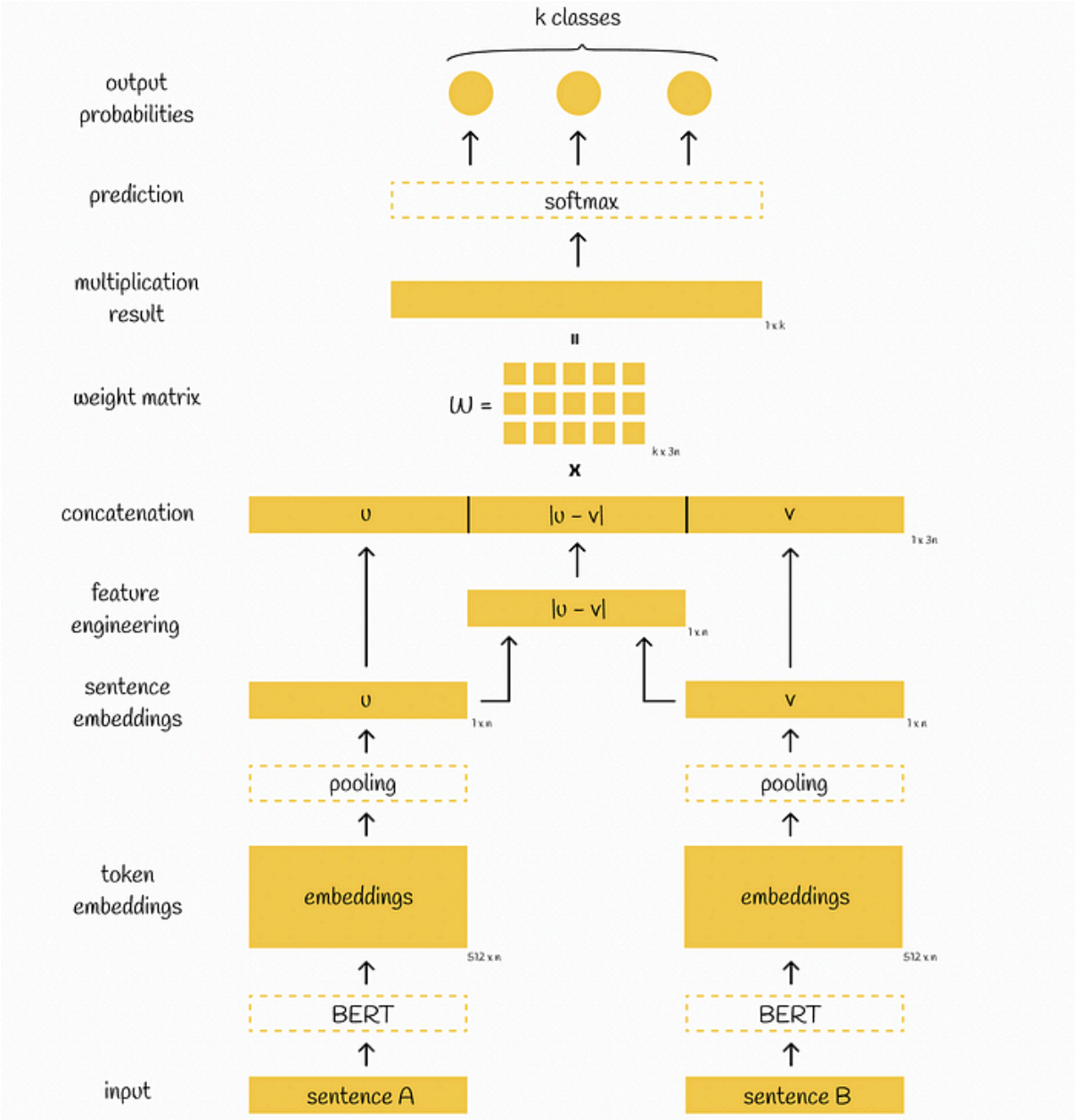
Deriving independent sentence embeddings is one of the main problems of BERT. To alleviate this aspect, SBERT was developed.

We explained the cross-encoder architecture for sentence similarity with BERT. SBERT is similar but drops the final classification head, and processes one sentence at a time. SBERT then uses mean pooling on the final output layer to produce a sentence embedding.
Unlike BERT, SBERT is fine-tuned on sentence pairs using a siamese architecture. We can think of this as having two identical BERTs in parallel that share the exact same network weights.



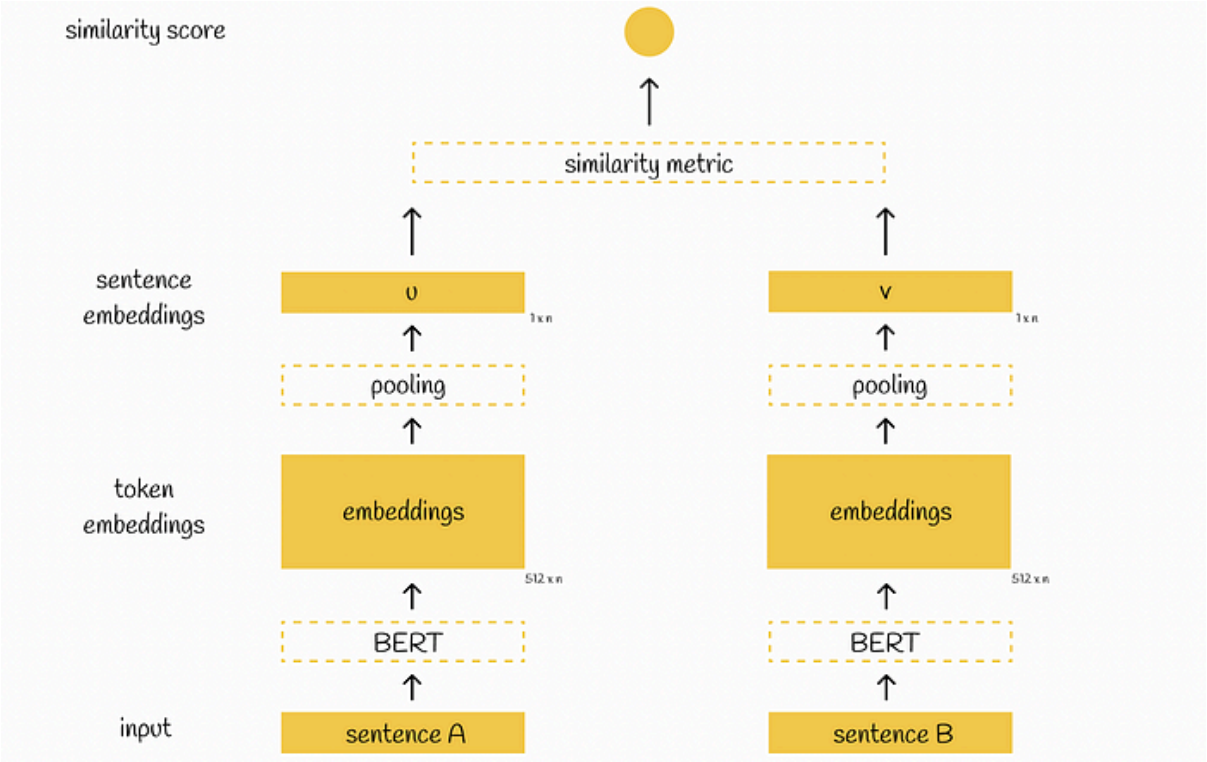
In reality, we are using a single BERT model. However, because we process sentence A followed by sentence B as pairs during training, it is easier to think of this as two models with tied weights.

SBERT Objective Functions
By using these two vectors u and v, three approaches for optimizing different objectives are discussed below.
4.1. Classification
The three vectors u, v and |u-v| are concatenated, multiplied by a trainable weight matrix W and the multiplication result is fed into the softmax classifier which outputs normalized probabilities of sentences corresponding to different classes. The cross-entropy loss function is used to update the weights of the model.



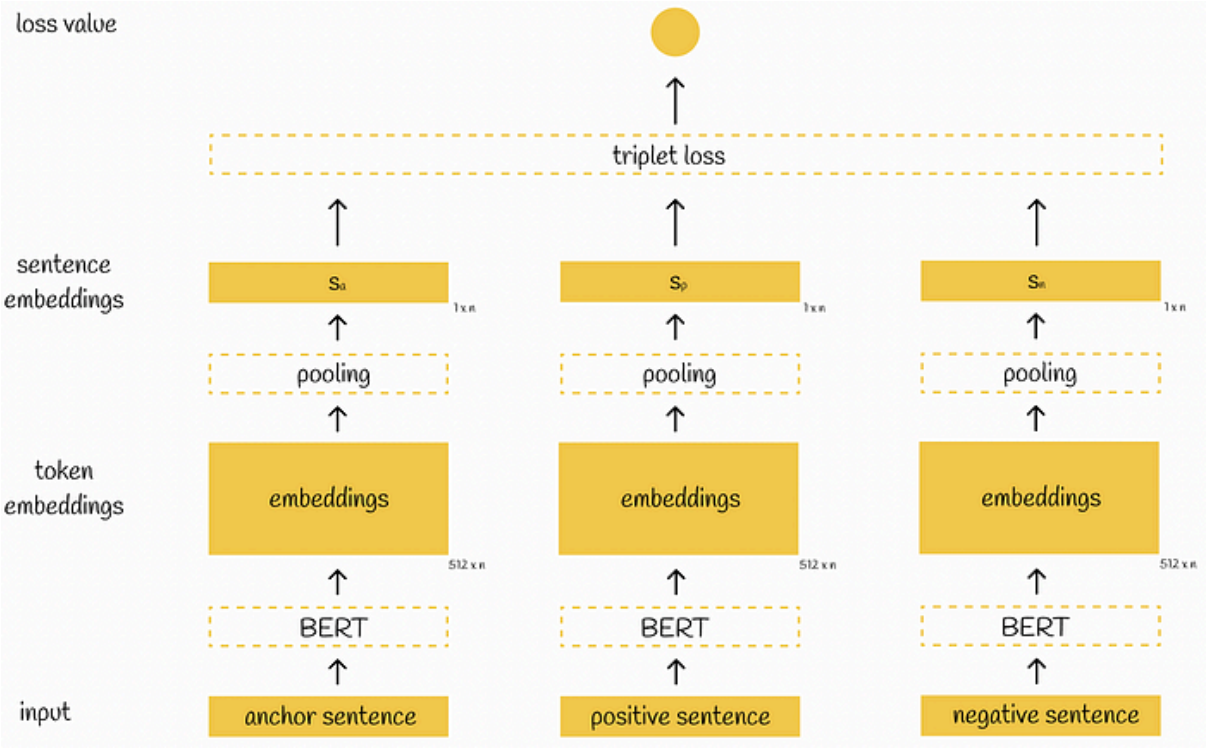
4.2. Regression

In this formulation, after getting vectors u and v, the similarity score between them is directly computed by a chosen similarity metric. The predicted similarity score is compared with the true value and the model is updated by using the MSE loss function.



4.3. Triplet Loss

The triplet objective introduces a triplet loss which is calculated on three sentences usually named anchor, positive and negative. It is assumed that anchor and positive sentences are very close to each other while anchor and negative are very different. During the training process, the model evaluates how closer the pair (anchor, positive) is, compared to the pair (anchor, negative).

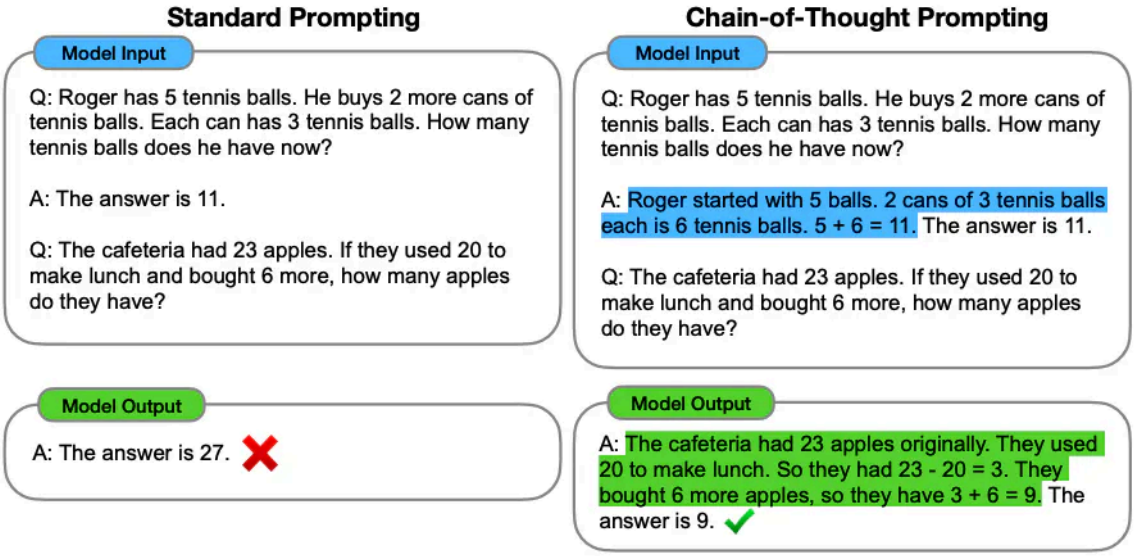


Zero Shot, Few Shot, Chain of thought

Zero Shot
Zero-shot prompting means that the prompt used to interact with the model won't contain examples or demonstrations. The zero-shot prompt directly instructs the model to perform a task without any additional examples to steer it.Instruction tuning has been shown to improve zero-shot learning. Instruction tuning is essentially the concept of fine tuning models on datasets described via instructions.

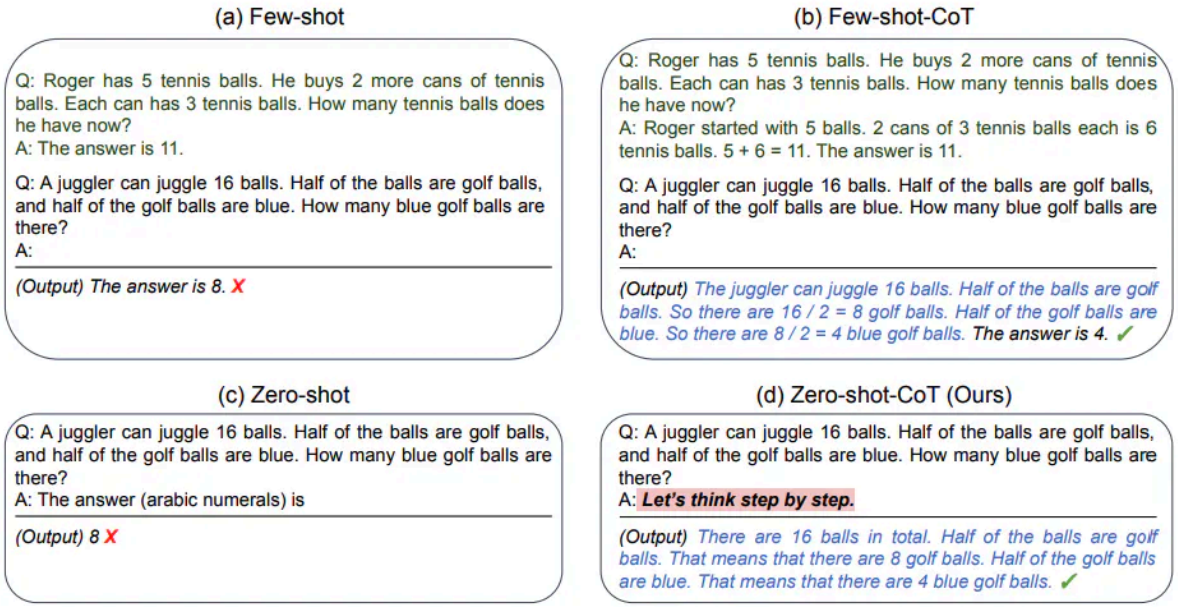
Few Shot
While large-language models demonstrate remarkable zero-shot capabilities, they still fall short on more complex tasks when using the zero-shot setting. Few-shot prompting can be used as a technique to enable in-context learning where we provide demonstrations in the prompt to steer the model to better performance. The demonstrations serve as conditioning for subsequent examples where we would like the model to generate a response.
Standard few-shot prompting works well for many tasks but is still not a perfect technique, especially when dealing with more complex reasoning tasks.

Chain of thought



Chain-of-thought (CoT) prompting enables complex reasoning capabilities through intermediate reasoning steps. You can combine it with few-shot prompting to get better results on more complex tasks that require reasoning before responding.

Zero Shot CoT
One recent idea that came out more recently is the idea that essentially involves adding "Let's think step by step" to the original prompt.



Prompt Chaining

To improve the reliability and performance of LLMs, one of the important prompt engineering techniques is to break tasks into its subtasks. Once those subtasks have been identified, the LLM is prompted with a subtask and then its response is used as input to another prompt. This is what's referred to as prompt chaining, where a task is split into subtasks with the idea to create a chain of prompt operations.

Prompt chaining is useful to accomplish complex tasks which an LLM might struggle to address if prompted with a very detailed prompt. In prompt chaining, chain prompts perform transformations or additional processes on the generated responses before reaching a final desired state.

Besides achieving better performance, prompt chaining helps to boost the transparency of your LLM application, increases controllability, and reliability. This means that you can debug problems with model responses much more easily and analyze and improve performance in the different stages that need improvement.

Prompt chaining is particularly useful when building LLM-powered conversational assistants and improving the personalization and user experience of your applications.

Retrieval Augmented Generation

Introduction:

- RAG is an AI framework that improves the accuracy and reliability of large language models (LLMs) by grounding them in external knowledge bases.
- LLMs can be inconsistent and prone to errors, lacking true understanding of word meaning.
- RAG addresses these issues by providing access to up-to-date facts and verifiable sources, increasing user trust.

Purpose of RAG:

- Grounding LLMs on external knowledge for improved responses.

- Overcoming inconsistencies in LLM-generated answers.

Challenges Addressed by RAG:

- Inconsistency in LLM responses.
- Lack of understanding of the meaning of words by LLMs.
- Reduction of opportunities for the model to leak sensitive data.

Benefits of RAG:

Accuracy and Fact-Checking:

Ensures LLM responses are based on reliable sources, allowing users to verify claims.

Reduced Bias and Hallucination:

Limits LLM reliance on internal biases and prevents fabrication of information.

Lower Cost and Maintenance:

Reduces the need for continuous LLM retraining and updates, saving computational resources.

How RAG Works:

RAG consists of two distinct phases: retrieval and content generation. In the retrieval phase, algorithms search for and retrieve relevant information from external knowledge bases. This information is then used in the generative phase, where the LLM synthesizes an answer based on both the augmented prompt and its internal representation of training data.

Phase 1: Retrieval

- Relevant information is retrieved from external sources based on the user's prompt or question.
- Sources vary depending on the context (open-domain internet vs. closed-domain enterprise data).

Phase 2: Content Generation

- The retrieved information is appended to the user's prompt and fed to the LLM.
- The LLM generates a personalized answer based on the augmented prompt and its internal knowledge base.
- The answer can be delivered with links to its sources for transparency.

Advantages and Applications of RAG

RAG offers several advantages, including access to the latest, reliable facts, reduction in sensitive data leakage and decreased need for continuous model retraining. It finds applications in personalized responses, verifiable answers and lowering computational and financial costs in enterprise settings.

1. Access to current and reliable information.
2. Reduced opportunities for sensitive data leakage.
3. Lower computational and financial costs in LLM-powered applications.

Implementation and Workflows

RAG operates in an “open book” manner, allowing LLMs to respond to questions by browsing through external content. It involves a retrieval phase, where relevant information is gathered and a generative phase, where the LLM synthesizes a response using both external and internal knowledge.

Open-Book Approach:

- Contrasted with closed-book exams, where LLMs rely on memory.
- Model’s response involves browsing through external content.

Workflow:

- Retrieval phase: Search and gather external information.

- Generative phase: Synthesize a personalized answer using internal and external knowledge.

Teaching LLMs to Recognize Limitations

LLMs, when faced with ambiguous or complex queries, may provide inaccurate responses. RAG helps in training LLMs to recognize unanswerable questions and prompts them to either admit uncertainty or ask clarifying questions.

Recognition of Limitations:

- LLMs prone to making things up in challenging scenarios.
- Training LLMs to explicitly recognize unanswerable questions.

Challenges and Ongoing Innovations in Retrieval-Augmented Generation

While RAG is a powerful tool, there are still some challenges persist and ongoing innovations are necessary. Lots of Research is focused on improving both the retrieval and generation ends of the process to enhance the effectiveness of RAG.

Challenges:

- Imperfections in RAG.
- Enriching prompts with relevant information using vectors.

Innovations and Research:

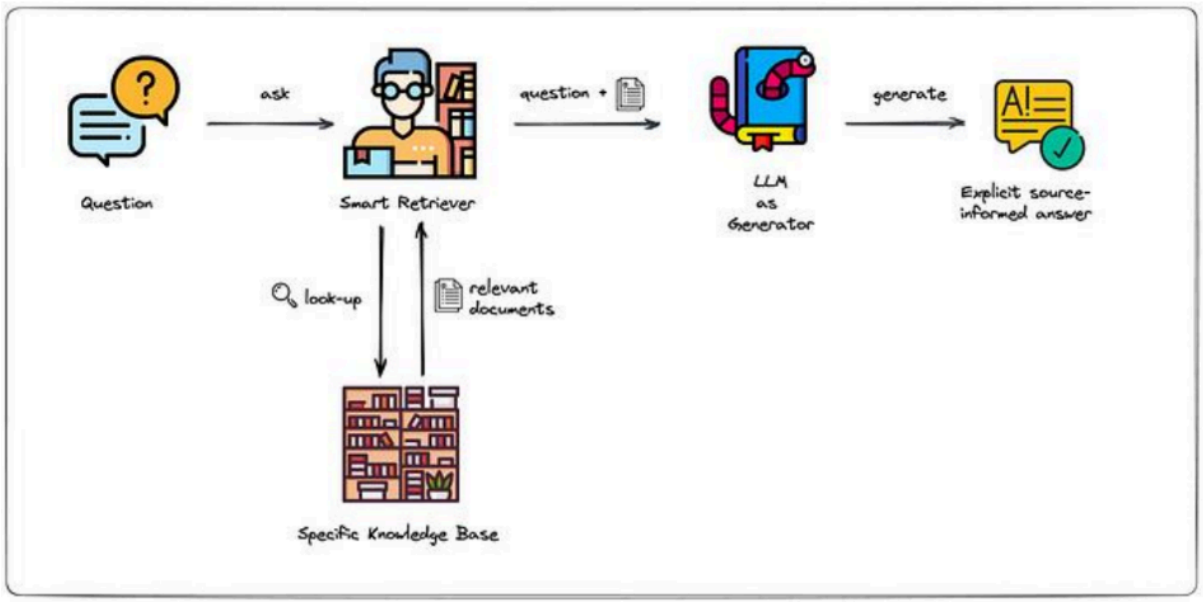
- Focus on retrieval: Finding and fetching the most relevant information.
- Focus on generation: Structuring information for richer responses.

Example Scenario: Customer Care Chatbot

- Alice, an employee, asks about taking vacation in half-day increments.
- The LLM retrieves relevant data from Alice's HR files and company policies.
- It generates a personalized answer confirming her vacation allowance and half-day eligibility.
- The answer is delivered with links to the HR files and policies for verification.

Challenges and Future Directions

- Training LLMs to recognize "unknowns" and avoid making things up.
- Improving retrieval algorithms for finding the most relevant information.
- Optimizing generation techniques for incorporating retrieved information effectively.



Method	Definition	Primary use case	Data requirements	Advantages	Considerations
Prompt engineering	Crafting specialized prompts to guide LLM behavior	Quick, on-the-fly model guidance	None	Fast, cost-effective, no training required	Less control than fine-tuning
Retrieval augmented generation (RAG)	Combining an LLM with external knowledge retrieval	Dynamic datasets and external knowledge	External knowledge base or database (e.g., vector database)	Dynamically updated context, enhanced accuracy	Increases prompt length and inference computation
Fine-tuning	Adapting a pretrained LLM to specific datasets or domains	Domain or task specialization	Thousands of domain-specific or instruction examples	Granular control, high specialization	Requires labeled data, computational cost
Pretraining	Training an LLM from scratch	Unique tasks or domain-specific corpora	Large datasets (billions to trillions of tokens)	Maximum control, tailored for specific needs	Extremely resource-intensive

Prompt engineering

Retrieval augmented generation (RAG)

Fine-tuning

Pre-train from scratch

Complexity/Compute-intensiveness

Tokenizers

Tokenization, in the realm of Natural Language Processing (NLP) and machine learning, refers to the process of converting a sequence of text into smaller parts, known as tokens.

These tokens can be as small as characters or as long as words. The primary reason this process matters is that it helps machines understand human language by breaking it down into bite-sized pieces, which are easier to analyze.

Imagine you're trying to teach a child to read. Instead of diving straight into complex paragraphs, you'd start by introducing them to individual letters, then syllables, and finally, whole words. In a similar vein, tokenization breaks down vast stretches of text into more digestible and understandable units for machines.
The primary goal of tokenization is to represent text in a manner that's meaningful for machines without losing its context. By converting text into tokens, algorithms can more easily identify patterns. This pattern recognition is crucial because it makes it possible for machines to understand and respond to human input. For instance, when a machine encounters the word "running", it doesn't see it as a singular entity but rather as a combination of tokens that it can analyze and derive meaning from.
To delve deeper into the mechanics, consider the sentence, "Chatbots are helpful." When we tokenize this sentence by words, it transforms into an array of individual words: ["Chatbots", "are", "helpful"].
This is a straightforward approach where spaces typically dictate the boundaries of tokens. However, if we were to tokenize by characters, the sentence would fragment into: ["C", "h", "a", "t", "b", "o", "t", "s", " ", "a", "r", "e", " ", "h", "e", "l", "p", "f", "u", "l"].
This character-level breakdown is more granular and can be especially useful for certain languages or specific NLP tasks.
In essence, tokenization is akin to dissecting a sentence to understand its anatomy. Just as doctors study individual cells to understand an organ, NLP practitioners use tokenization to dissect and understand the structure and meaning of text.
It's worth noting that while our discussion centers on tokenization in the context of language processing, the term "tokenization" is also used in the realms of security and privacy, particularly in data protection practices like credit card tokenization. In such scenarios, sensitive data elements are replaced with non-sensitive equivalents, called tokens. This distinction is crucial to prevent any confusion between the two contexts.
Creating Vocabulary is the ultimate goal of Tokenization.
One of the simplest hacks to boost the performance of the NLP model is to create a vocabulary out of top K frequently occurring words.

Which Tokenization Should you use?

As discussed earlier, tokenization can be performed on word, character, or subword level.

It’s a common question – which Tokenization should we use while solving an NLP task? Let’s

address this question here.

Word Tokenization

Word Tokenization is the most commonly used tokenization algorithm. It splits a piece of text into individual words based on a certain delimiter. Depending upon delimiters, different word-level tokens are formed. Pretrained Word Embeddings such as Word2Vec and GloVe comes under word tokenization.
But, there are few drawbacks to this.
Drawbacks of Word Tokenization
One of the major issues with word tokens is dealing with Out Of Vocabulary (OOV) words. OOV words refer to the new words which are encountered at testing. These new words do not exist in the vocabulary. Hence, these methods fail in handling OOV words.
But wait – don't jump to any conclusions yet!

- A small trick can rescue word tokenizers from OOV words. The trick is to form the vocabulary with the Top K Frequent Words and replace the rare words in training data with unknown tokens (UNK). This helps the model to learn the representation of OOV words in terms of UNK tokens
- So, during test time, any word that is not present in the vocabulary will be mapped to a UNK token. This is how we can tackle the problem of OOV in word tokenizers.
- The problem with this approach is that the entire information of the word is lost as we are mapping OOV to UNK tokens. The structure of the word might be helpful in representing the word accurately. And another issue is that every OOV word gets the same representation
- Another issue with word tokens is connected to the size of the vocabulary. Generally, pre-trained models are trained on a large volume of the text corpus. So, just imagine building the vocabulary with all the unique words in such a large corpus. This explodes the vocabulary! This opens the door to Character Tokenization.

Character Tokenization

Character Tokenization splits apiece of text into a set of characters. It overcomes the drawbacks we saw above about Word Tokenization.

- Character Tokenizers handles OOV words coherently by preserving the information of the word. It breaks down the OOV word into characters and represents the word in terms of these characters
- It also limits the size of the vocabulary. Want to talk a guess on the size of the vocabulary? 26 since the vocabulary contains a unique set of characters

Drawbacks of Character Tokenization
Character tokens solve the OOV problem but the length of the input and output sentences increases rapidly as we are representing a sentence as a sequence of characters. As a result, it becomes challenging to learn the relationship between the characters to form meaningful words.
This brings us to another tokenization known as Subword Tokenization which is in between a Word and Character tokenization.

Subword Tokenization

Subword Tokenization splits the piece of text into subwords (or n-gram characters). For example, words like lower can be segmented as low-er, smartest as smart-est, and so on.

Transformed based models – the SOTA in NLP – rely on Subword Tokenization algorithms for preparing vocabulary. Now, I will discuss one of the most popular Subword Tokenization algorithm known as Byte Pair Encoding (BPE).

Byte Pair Encoding (BPE)

Byte Pair Encoding (BPE) is a widely used tokenization method among transformer-based models. BPE addresses the issues of Word and Character Tokenizers:

- BPE tackles OOV effectively. It segments OOV as subwords and represents the word in terms of these subwords
- The length of input and output sentences after BPE are shorter compared to character tokenization

BPE is a word segmentation algorithm that merges the most frequently occurring character or character sequences iteratively. Here is a step by step guide to learn BPE.
Steps to learn BPE

1. Split the words in the corpus into characters after appending </w>
2. Initialize the vocabulary with unique characters in the corpus
3. Compute the frequency of a pair of characters or character sequences in corpus
4. Merge the most frequent pair in corpus
5. Save the best pair to the vocabulary
6. Repeat steps 3 to 5 for a certain number of iterations

We will understand the steps with an example.

Corpus

low	lower	newest
low	lower	newest
low	widest	newest
low	widest	newest
low	widest	newest

Consider a corpus:
1a) Append the end of the word (say </w>) symbol to every word in the corpus:

Corpus

low</w>	lower</w>	newest</w>
low</w>	lower</w>	newest</w>
low</w>	widest</w>	newest</w>
low</w>	widest</w>	newest</w>
low</w>	widest</w>	newest</w>

1b) Tokenize words in a corpus into characters:

Corpus

l o w </w>	l o w e r </w>	n e w e s t </w>
l o w </w>	l o w e r </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>
l o w </w>	w i d e s t</w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>

2. Initialize the vocabulary:

Vocabulary

d	e	i	l	n	o	s	t	w

Iteration 1:

3. Compute frequency:

Frequency

d-e (3)	l-o (7)	t-</w> (8)
e-r (2)	n-e (5)	w-</w> (5)
e-s (8)	o-w (7)	w-e (7)
e-w (5)	r-</w> (2)	w-i (3)
i-d (3)	s-t (8)	

4. Merge the most frequent pair:

Corpus

l o w </w>	l o w e r </w>	n e w es t </w>
l o w </w>	l o w e r </w>	n e w es t </w>
l o w </w>	w i d es t </w>	n e w es t </w>
l o w </w>	w i d es t</w>	n e w es t </w>
l o w </w>	w i d es t </w>	n e w es t </w>

5. Save the best pair:

Vocabulary

d	e	i	l	n	o	s	t	w
es								

Repeat steps 3-5 for every iteration from now. Let me illustrate for one more iteration.
Iteration 2:

3. Compute frequency:

Frequency

d-es (3)	l-o (7)	w-</w> (5)
e-r (2)	n-e (5)	w-es (5)
e-w (5)	o-w (7)	w-e (2)
es-t (8)	r-</w> (2)	w-i (3)
i-d (3)	t-</w> (8)	

4. Merge the most frequent pair:

Corpus

l o w </w>	l o w e r </w>	n e w est </w>
l o w </w>	l o w e r </w>	n e w est </w>
l o w </w>	w i d est </w>	n e w est </w>
l o w </w>	w i d est </w>	n e w est </w>
l o w </w>	w i d est </w>	n e w est </w>

5. Save the best pair:

Vocabulary

d	e	i	l	n	o	s	t	w
es	est							

After 10 iterations, BPE merge operations looks like:

Vocabulary

d	e	i	l	n	o	s	t	w
es	est	est</w>	lo	low	low</w>	ne	new	newest</w>

Pretty straightforward, right?
Applying BPE to OOV words
But, how can we represent the OOV word at test time using BPE learned operations? Any ideas? Let's answer this question now.
At test time, the OOV word is split into sequences of characters. Then the learned operations are applied to merge the characters into larger known symbols.
– Neural Machine Translation of Rare Words with Subword Units, 2016
Here is a step by step procedure for representing OOV words:

1. Split the OOV word into characters after appending </w>
2. Compute pair of character or character sequences in a word
3. Select the pairs present in the learned operations
4. Merge the most frequent pair
5. Repeat steps 2 and 3 until merging is possible

Byte-Level Byte-Pair Encoding (BPE)

An extension of the original BPE, Byte-Level BPE operates on a byte-level rather than character-level. It encodes each token as a sequence of bytes rather than characters. This allows it to:

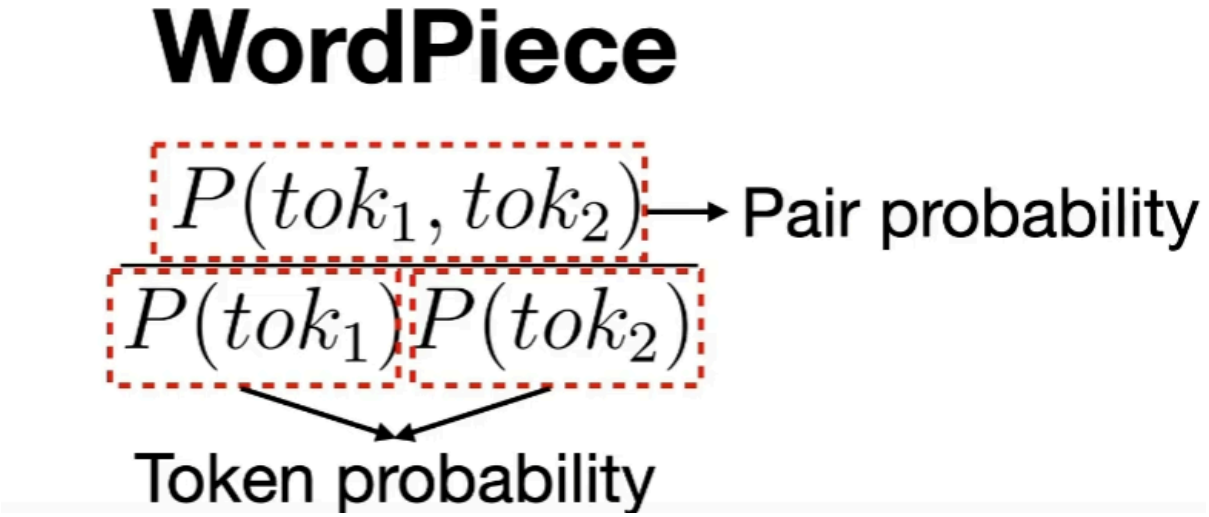
- Better handle Unicode characters and multi-lingual text
- Avoid maintaining separate vocabularies for each language
- Achieve open-vocabulary by representing any unseen word as a sequence of subword tokens

Byte-Level BPE is used by models like GPT-2 for text generation.

Word Piece Tokenization

BPE takes a pair of tokens (bytes), looks at the frequency of each pair, and merges the pair which has the highest combined frequency. The process is greedy as it looks for the highest combined frequency at each step.
So, what's the problem with BPE? It can have instances where there is more than one way to encode a particular word. It then gets difficult for the algorithm to choose subword tokens as there is no way to prioritize which one to use first. Hence, the same input can be represented by different encodings impacting the accuracy of the learned representations.

The only difference between WordPiece and BPE is the way in which symbol pairs are added to the vocabulary. At each iterative step, WordPiece chooses a symbol pair which will result in the largest increase in likelihood upon merging. Maximizing the likelihood of the training data is equivalent to finding the symbol pair whose probability divided by the probability of the first followed by the probability of the second symbol in the pair is greater than any other symbol pair.

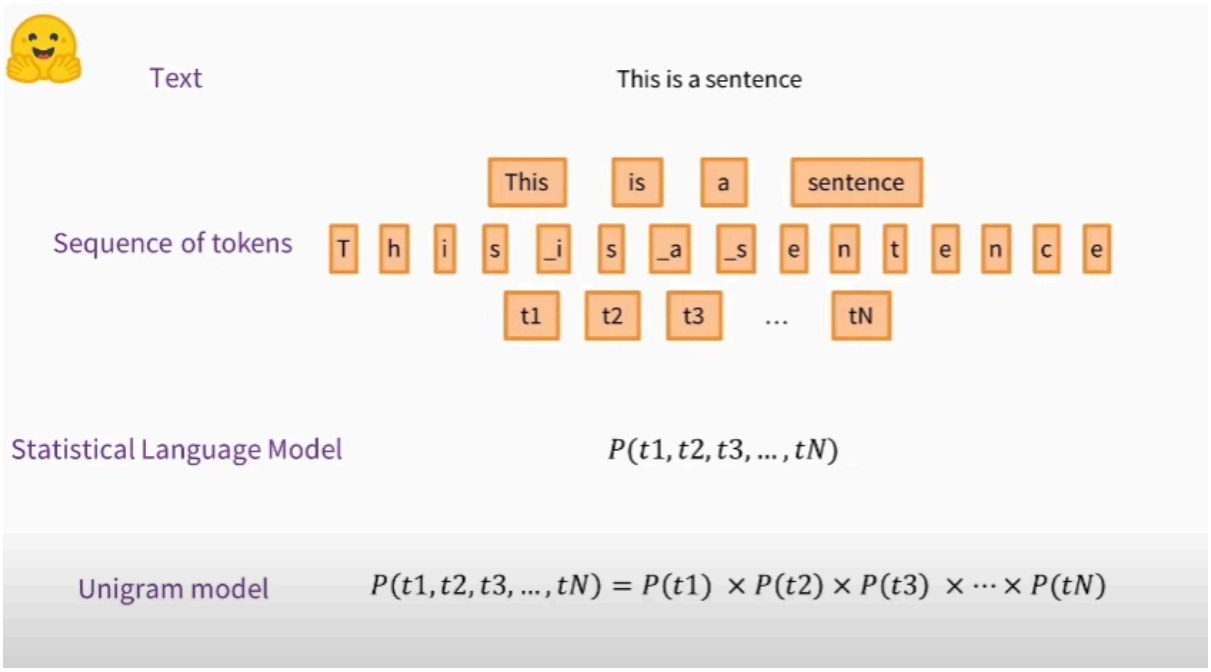


Sentence Piece Tokenization

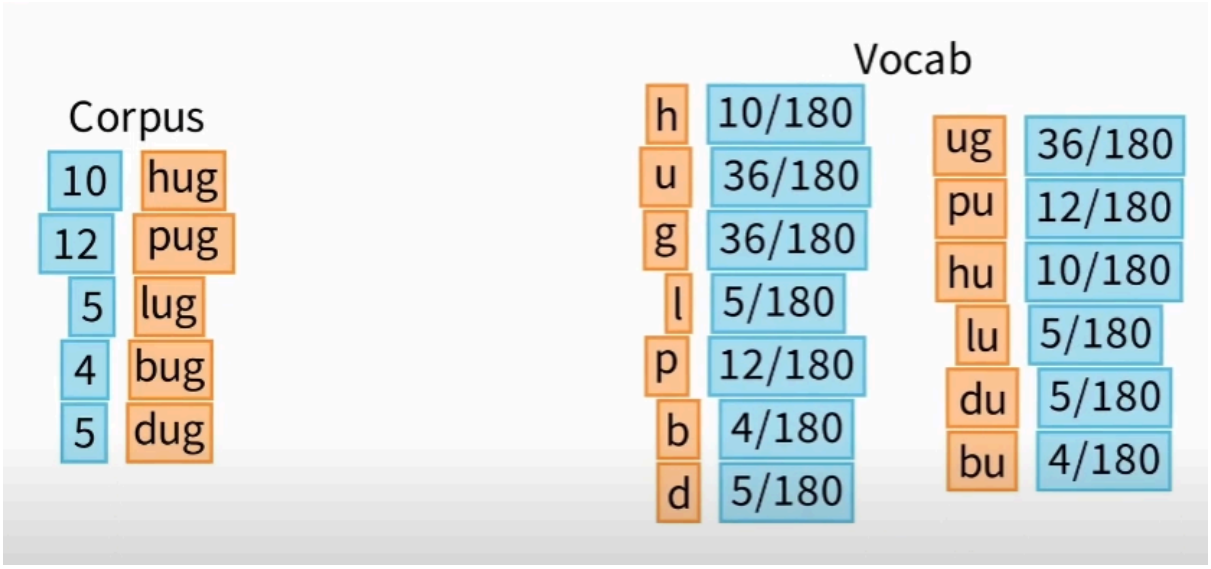
1. Initialize the unigram probabilities. Remember P(x) = P(x_1)…P(x_n) so once we have the unigrams, we have the probability of any sequence. In our code, we just use the BPE frequency counts to get closer to the objective.
2. M-step: compute the most probable unigram sequence given the current probabilities. This defines a single tokenization. Implementing this requires some thought.
3. E-step: given the current tokenization, recompute the unigram probabilities by counting the occurrence of all subwords in the tokenization. The frequentist unigram probability is just the frequency with which that unigram occurs. In practice, it is of no more difficulty to Bayesian-ify this (see the Appendix) and instead compute

$$P_{\text{unigram}} = \frac{c_i}{\sum_{i=1}^{M} c_i} \rightarrow \frac{e^{\Psi(c_i)}}{e^{\Psi(\sum_{i=1}^{M} c_i)}}$$

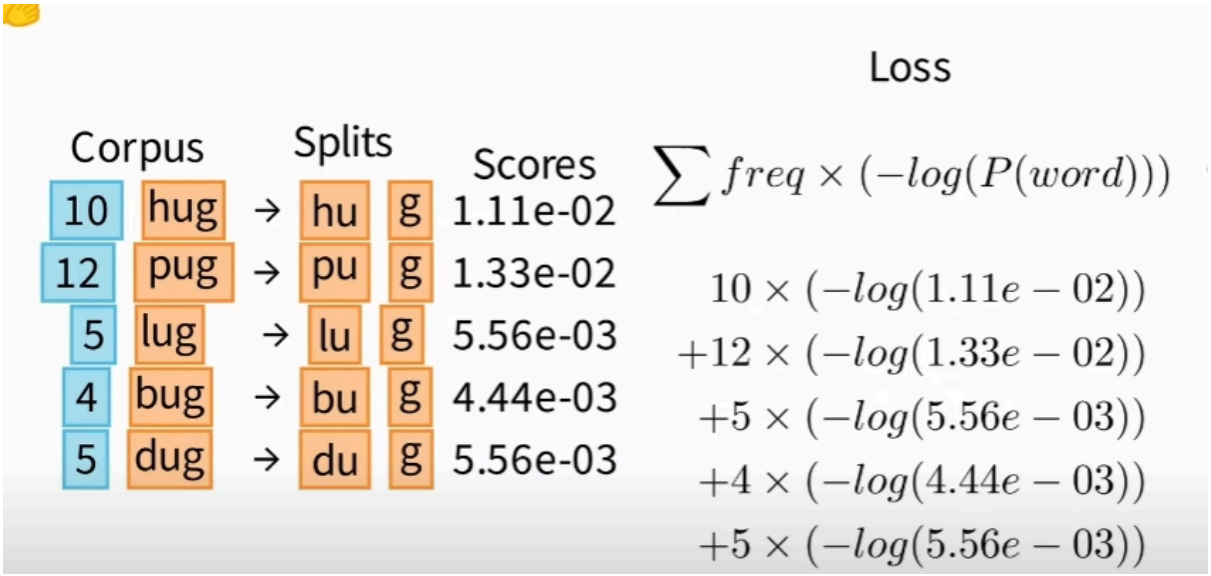
Here, c_i is the count of subword (unigram) i in the current tokenization. M is the total number of subwords. Psi is the digamma function. The arrow indicates how we Bayesian-ify.
4. Repeat steps 2 and 3 until convergence. The log-likelihood is theoretically guaranteed to monotonically increase, so if it doesn't you've got a bug.



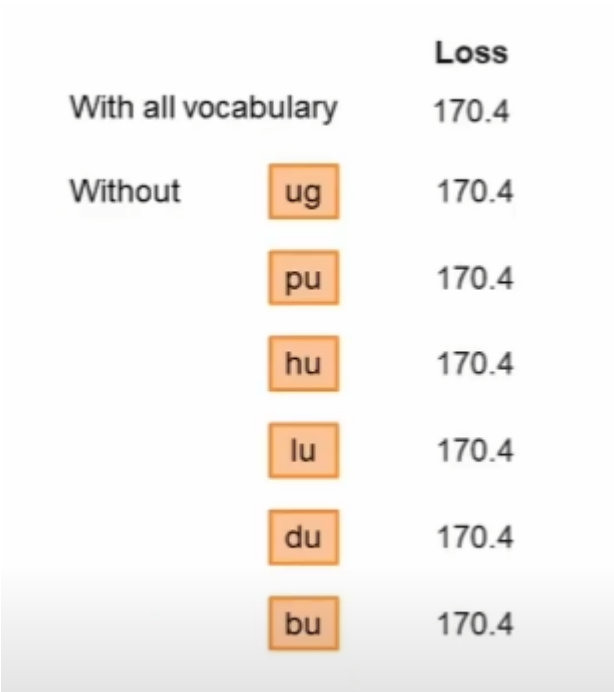
Iteration-1



Choose the max one if tied then choose randomly



M-step: Remove the token that least impacts the loss on the corpus



2nd iteration

E-step: Estimate the probabilities

M-step: Remove the token that least impacts the loss on the corpus

Causal Models vs Masked LM

Causal Language Modeling (CLM)

CLM is an autoregressive method where the model is trained to predict the next token in a sequence given the previous tokens. CLM is used in models like GPT-2 and GPT-3 and is well-suited for tasks such as text generation and summarization. However, CLM models have unidirectional context, meaning they only consider the past and not the future context when generating predictions.

Masked Language Modeling (MLM)

MLM is a training method used in models like BERT, where some tokens in the input sequence are masked, and the model learns to predict the masked tokens based on the surrounding context. MLM has the advantage of bidirectional context, allowing the model to consider both past and future tokens when making predictions. This approach is especially useful for tasks like text classification, sentiment analysis, and named entity recognition.

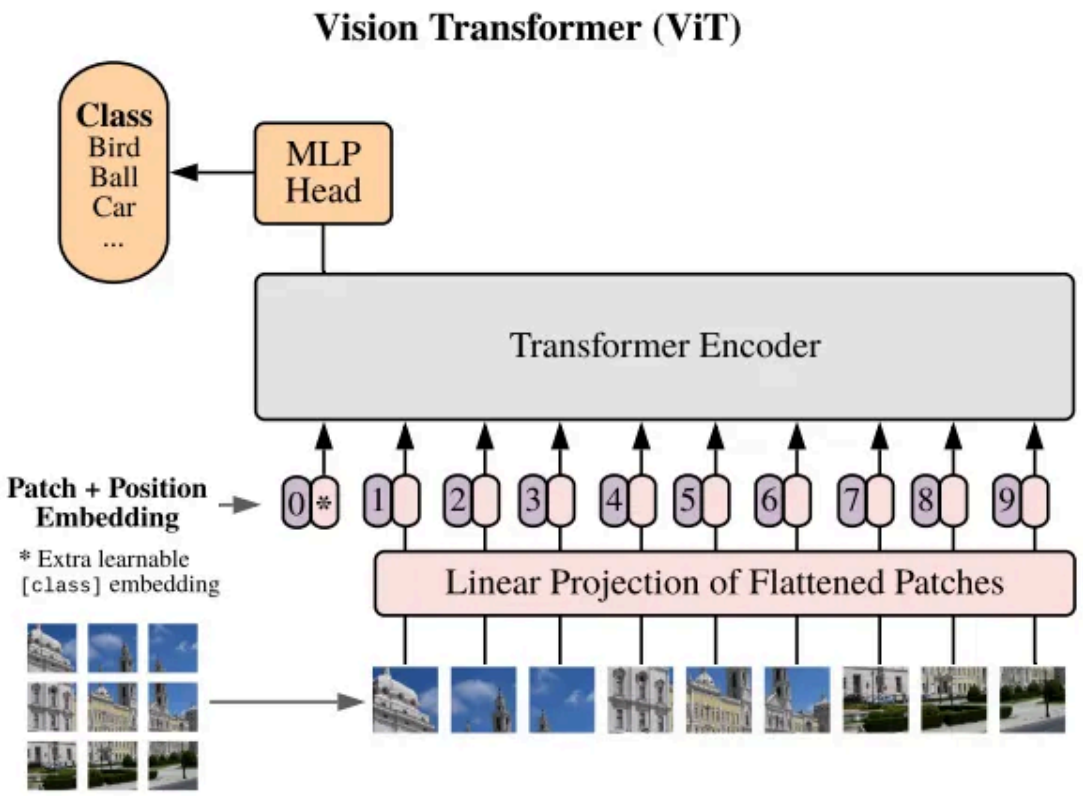
Sequence-to-Sequence (Seq2Seq)

Seq2Seq models consist of an encoder-decoder architecture, where the encoder processes the input sequence and the decoder generates the output sequence. This approach is commonly used in tasks like machine translation, summarization, and question-answering. Seq2Seq models can handle more complex tasks that involve input-output transformations, making them versatile for a wide range of NLP tasks.

Prefix Language Modeling (Prefix LM)

In this task, a separate ‘prefix’ section is separated from the main sequence. Within the prefix, any token can attend to any other token (non-causal). Outside of the prefix, decoding proceeds autoregressively.

ViT

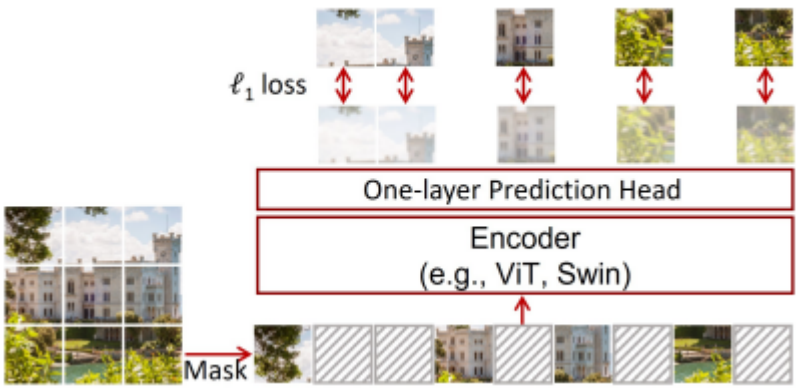


Issue with ViT is it needs large data to pretrain it in supervised fashion. There is no unsupervised pretraining.

Remedy

Masked Image Modeling Analogous to Masked Language Modeling in NLP.

Image patches are masked randomly and the Transformer should predict the masked patch in the pretraining step. Later the trained transformer can be finetuned on a downstream supervised task.

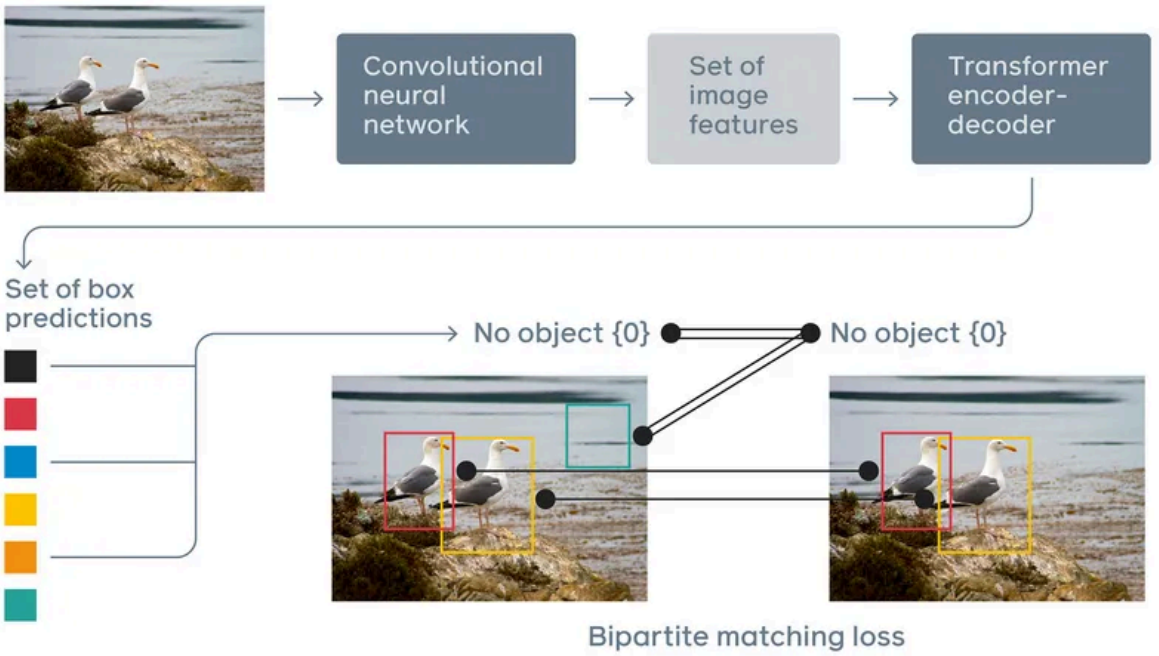


DETR

In DETR, object detection problem is modeled as a direct set prediction problem.
The approach don't require hand crafted algorithms like non-maximum suppression procedure or anchor generation that explicitly encode our prior knowledge about the task. It makes the detection pipeline a simple end to end unified architecture.
The two novel components of the new framework, called DEtection TRansformer or DETR, are

- a set-based global loss that forces unique predictions via bipartite matching.
- a transformer encoder-decoder architecture.

Given a fixed small set of learned object queries, DETR reasons about the relations of the objects and the global image context to directly output the final set of predictions in parallel.
How DETR differs from other object detection methods?
DETR formulates the object detection task as an image-to-set problem.
Given an image, the model must predict an unordered set (or list) of all the objects present, each represented by its class, along with a tight bounding box surrounding each one.
Transformer acts as a reasoning agent between the image features and the prediction.



DETR directly predicts (in parallel) the final set of detections by combining a common CNN with a Transformer architecture. During training, bipartite matching uniquely assigns predictions with ground truth boxes. Predictions with no match should yield a "no object" class prediction.

Inference

1. Calculate image features from a backbone.
2. To transformer encoder-decoder architecture.
3. Calculate set of predictions (all at once)

Training

1. Calculate image features from a backbone.
2. To transformer encoder-decoder architecture.
3. To a set loss function which performs bipartite matching between predicted and ground-truth objects to remove false or extra detection's.

Advantages of the DETR pipeline

- Easy to use
- No custom layers
- Easy extension to other tasks
- Aprior information about anchors or hand crafted algorithms like NMS are not needed

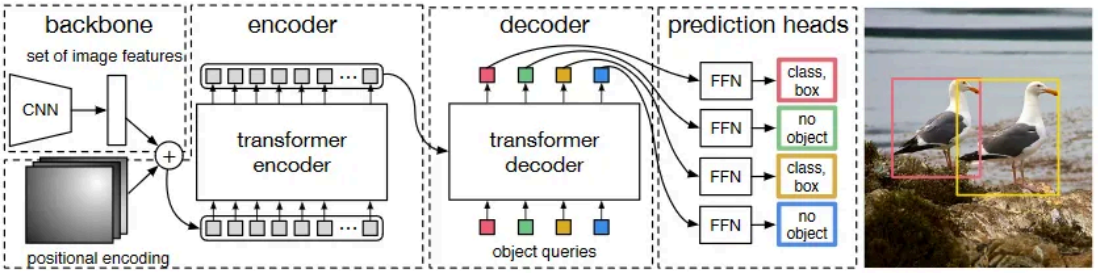


Fig. 2: DETR uses a conventional CNN backbone to learn a 2D representation of an input image. The model flattens it and supplements it with a positional encoding before passing it into a transformer encoder. A transformer decoder then takes as input a small fixed number of learned positional embeddings, which we call *object queries*, and additionally attends to the encoder output. We pass each output embedding of the decoder to a shared feed forward network (FFN) that predicts either a detection (class and bounding box) or a "`no object`" class.

Backbone
The authors have used ResNet50 as the backbone of DETR. Ideally, any backbone can be used depending upon the complexity of the task at hand
Backbones provide low dimensional representation of the image having refined features.

Encoder
First, a 1x1 convolution reduces the channel dimension of the high-level activation map from C to a smaller dimension d, creating a new feature map d×H×W. The encoder expects a sequence as input so it is collapsed to one dimension, resulting in a d×HW feature map.
Each encoder layer has a standard architecture and consists of a multi-head self-attention module and a feed forward network (FFN).
Since the transformer architecture is permutation-invariant, they supplement it with fixed positional encodings that are added to the input of each attention layer.

Decoder
The decoder follows the standard architecture of the transformer, transforming N embeddings of size d using multi-headed self- and encoder-decoder attention mechanisms. The difference with the original transformer is that DETR model decodes the N objects in parallel at each decoder layer.
These N input embeddings are learnt positional encodings that they refer to as object queries,and similarly to the encoder, they are added them to the input of each attention layer. The N object queries are transformed into an output embedding by the decoder. They are then independently decoded into box coordinates and class labels by a feed forward network (FFN), resulting N final predictions.
The decoder receives queries (initially set to zero), output positional encoding (object queries), and encoder memory, and produces the final set of predicted class labels and

bounding boxes through multiple multi-head self-attention and decoder-encoder attention. The first self-attention layer in the first decoder layer can be skipped.

Feed-Forward Network(FFN)

FNN is a 3-layer perceptron with ReLU activation function and hidden dimension d, and a linear projection layer. FFN layers are effectively multi-layer 1x1 convolutions, which have Md input and output channels.

The FFN predicts the normalized center coordinates, height and width of the box w.r.t. the input image, and the linear layer predicts the class label using a softmax function.

Since we predict a fixed-size set of N bounding boxes, where N is usually much larger than the actual number of objects of interest in an image, an additional special class label ∅is used to represent that no object is detected within a slot. This class plays a similar role to the "background" class in the standard object detection approaches.

Loss Function

The following explanation may seem a lot to grasp but trust me when you read it carefully it is just two simple steps.

1. Calculate the best match of predictions with respect to given ground truths using a graph technique with a cost function.(unique to DETR)
2. Next, we define a loss to penalize the class and box predictions. (usual step)

The loss function is an optimal bipartite matching function. Allow me to simplify it.

- Let y to be the number of ground truths in the image I.
- Let y_ be the number of predictions by the network.

The y_ is fixed to the value N which is assumed to be much larger than overall predictions in any image. The (actual predictions + the remaining ones are padded as no object labels) equates to N.

Next, they find a bipartite matching between these two sets using a matching function across a permutation of N elements with the lowest cost as follows:

$$\hat{\sigma} = \arg\min_{\sigma \in \mathfrak{S}_N} \sum_{i}^{N} \mathcal{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)}),$$

Fig 6 : Best match between pred and gt with lowest cost [1]

where Lmatch(yi,ˆyσ(i)) is a pair-wise matching cost between ground truth yi and a prediction with index σ(i). It is formulated as an assignment problem with m ground truth and n predictions and is computed efficiently with the Hungarian algorithm over mxn matrix.

$$-\mathbb{1}_{\{c_i \neq \varnothing\}} \hat{p}_{\sigma(i)}(c_i) + \mathbb{1}_{\{c_i \neq \varnothing\}} \mathcal{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)})$$

After receiving all matched pairs for the set, the next step is to compute the loss function, the Hungarian loss.

$$\mathcal{L}_{\text{Hungarian}}(y, \hat{y}) = \sum_{i=1}^{N} \left[-\log \hat{p}_{\hat{\sigma}(i)}(c_i) + \mathbb{1}_{\{c_i \neq \varnothing\}} \mathcal{L}_{\text{box}}(b_i, \hat{b}_{\hat{\sigma}}(i)) \right]$$

Box Loss Function (L_box)

The paper uses a linear combination of L1 and Generalized IOU loss (scale invariant in nature).

$$\lambda_{\text{iou}}\mathcal{L}_{\text{iou}}(b_i, \hat{b}_{\sigma(i)}) + \lambda_{\text{L1}}||b_i - \hat{b}_{\sigma(i)}||_1 \text{ where } \lambda_{\text{iou}}, \lambda_{\text{L1}} \in \mathbb{R} \text{ are hyperparameters.}$$

LayoutLM

LayoutLMv3 applies a unified text-image multimodal Transformer to learn cross-modal representations. The Transformer has a multilayer architecture and each layer mainly consists of multi-head self-attention and position-wise fully connected feed-forward networks. The input of Transformer is a concatenation of text embedding Y = y1:L and image embedding X = x1:M sequences, where L and M are sequence lengths for text and image respectively. Through the Transformer, the last layer outputs text-and-image contextual representations.

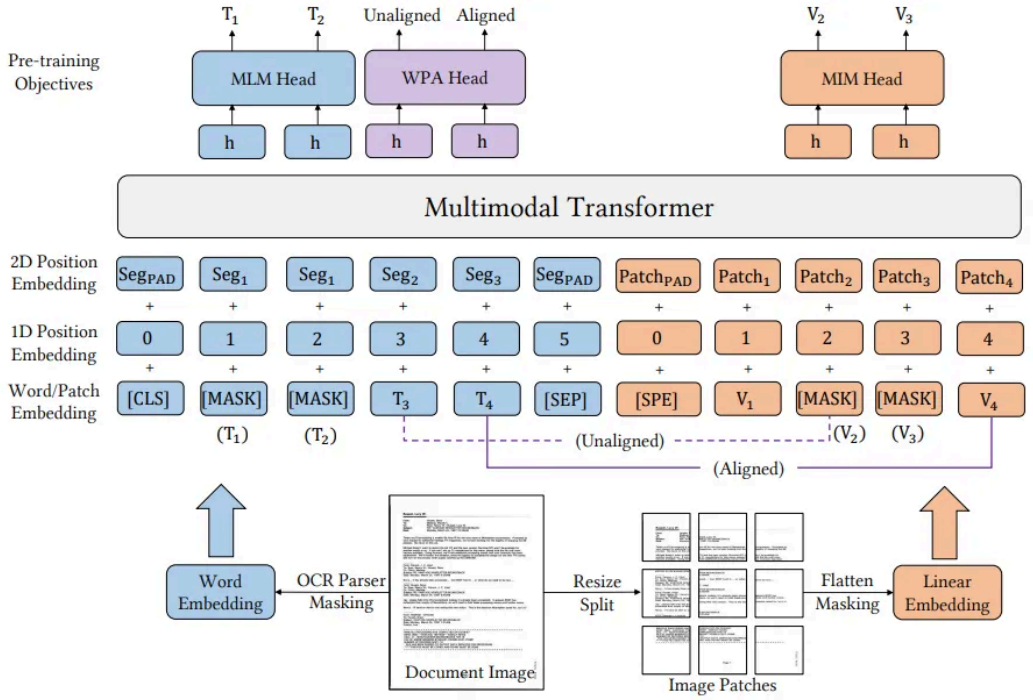


Figure 3: The architecture and pre-training objectives of LayoutLMv3. LayoutLMv3 is a pre-trained multimodal Transformer for Document AI with unified text and image masking objectives. Given an input document image and its corresponding text and layout position information, the model takes the linear projection of patches and word tokens as inputs and encodes them into contextualized vector representations. LayoutLMv3 is pre-trained with discrete token reconstructive objectives of Masked Language Modeling (MLM) and Masked Image Modeling (MIM). Additionally, LayoutLMv3 is pre-trained with a Word-Patch Alignment (WPA) objective to learn cross-modal alignment by predicting whether the corresponding image patch of a text word is masked. "Seg" denotes segment-level positions. "[CLS]", "[MASK]", "[SEP]" and "[SPE]" are special tokens.

Text Embedding

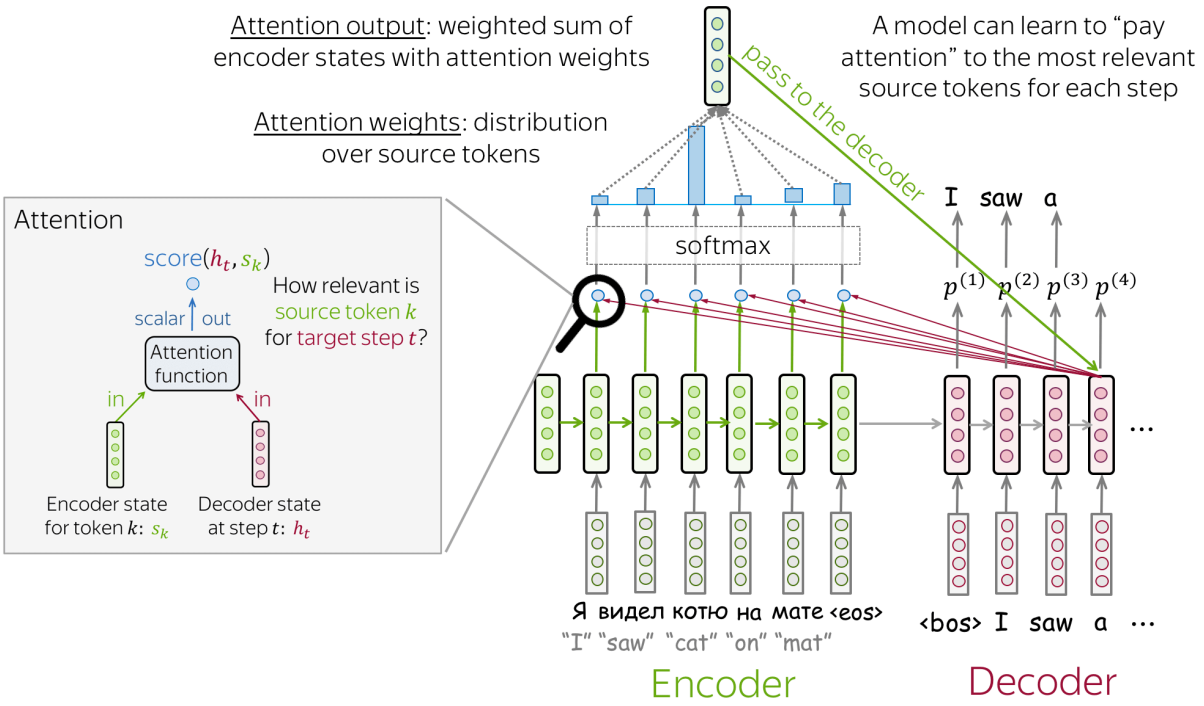
- Text embedding is a combination of word embeddings and position embeddings.
- The word Embeddings are initialized with a word embedding matrix from a pre-trained model RoBERTa.
- The position embeddings include 1D position and 2D layout position embeddings, where the 1D position refers to the index of tokens within the text sequence, and the 2D layout position refers to the bounding box coordinates of the text sequence.
- Following the LayoutLM, we normalize all coordinates by the size of images, and use embedding layers to embed x-axis, y-axis, width and height features separately.

- The LayoutLM and LayoutLMv2 adopt word-level layout positions, where each word has its positions. Instead, we adopt segment-level layout positions that words in a segment share the same 2D position since the words usually express the same semantic meaning.

Image Embedding
Document images ar represented with linear projection features of image patches before feeding them into the multimodal Transformer. A document image is resized into $H \times W$ and denote the image with I $\in$ R $C \times H \times W$, where C, H and W are the channel size, width and height of the image respectively. Image is split into a sequence of uniform $P \times P$ patches, linearly project the image patches to D dimensions and flatten them into a sequence of vectors, which length is $M = HW/P$ 2. Then learnable 1D position embeddings are added to each patch.

Pre Training
Masked Language Modeling (MLM): 30% of text tokens are masked with a span masking strategy with span lengths drawn from a Poisson distribution (λ = 3). The pre-training objective is to maximize the log-likelihood of the correct masked text tokens yl based on the contextual representations of corrupted sequences of image tokens XM' and text tokens YL', where M' and L' represent the masked positions. As the layout informationis kept unchanged, this objective facilitates the model to learn the correspondence between layout information and text and image context.
Masked Image Modelling (MIM): The MIM objective is a symmetry to the MLM objective, about 40% image tokens are randomly masked with the blockwise masking strategy. The MIM objective is driven by a cross-entropy loss to reconstruct the masked image tokens xm under the context of their surrounding text and image tokens. MIM facilitates learning high-level layout structures rather than noisy low-level details.
Word-Patch Alignment (WPA): The WPA objective is to predict whether the corresponding image patches of a text word are masked. Specifically, an aligned label is assigned to an unmasked text token when its corresponding image tokens are also unmasked. Otherwise, an unaligned label is assigned. The masked text tokens are excluded when calculating WPA loss to prevent the model from learning a correspondence between masked text words and image patches.

Seq2Seq with Attention

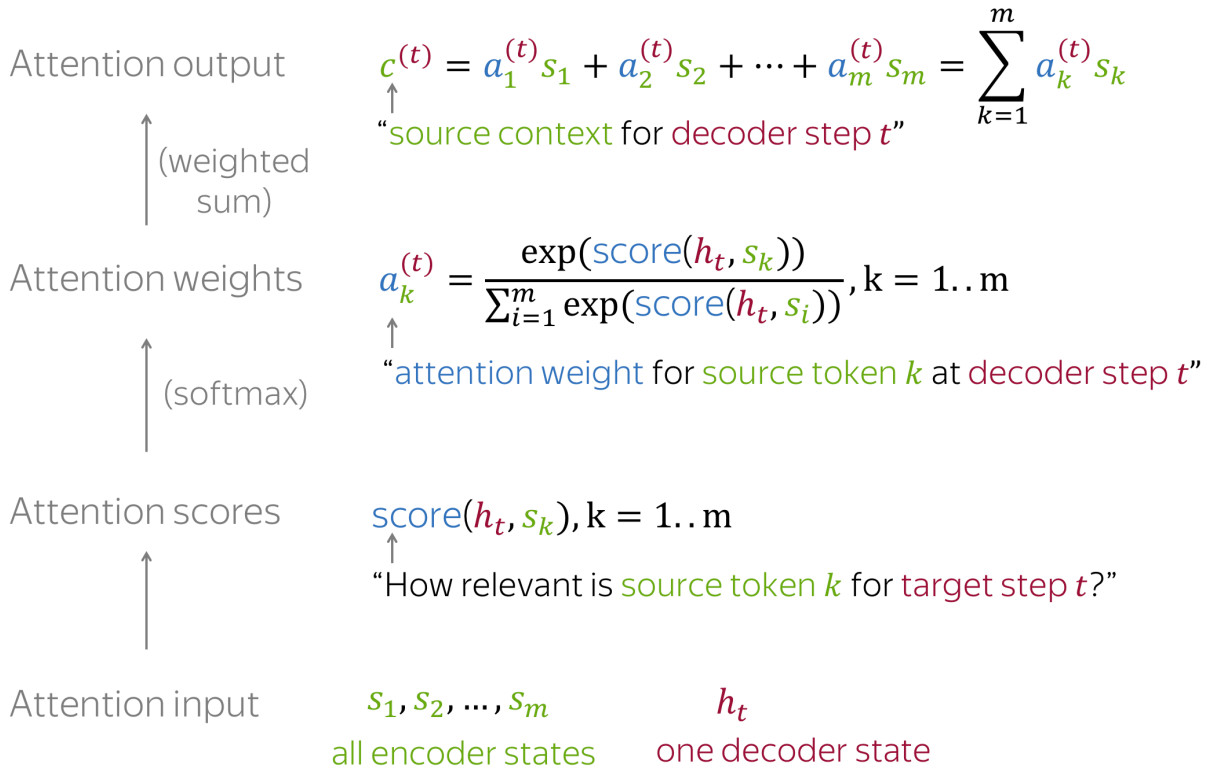


An attention mechanism is a part of a neural network. At each decoder step, it decides which source parts are more important. In this setting, the encoder does not have to compress the whole source into a single vector - it gives representations for all source tokens (for example, all RNN states instead of the last one).

At each decoder step, attention

- receives attention input: a decoder state ht and all encoder states $s1$, $s2$,..., sm;
- computes attention scores
  For each encoder state sk, attention computes its "relevance" for this decoder state ht. Formally, it applies an attention function which receives one decoder state and one encoder state and returns a scalar value $score(ht,sk)$;
- computes attention weights: a probability distribution - softmax applied to attention scores;
- computes attention output: the weighted sum of encoder states with attention weights.

The general computation scheme is shown below.



Note: Everything is differentiable - learned end-to-end!

The main idea that a network can learn which input parts are more important at each step. Since everything here is differentiable (attention function, softmax, and all the rest), a model with attention can be trained end-to-end. You don't need to specifically teach the model to pick the words you want - the model itself will learn to pick important information.

Attention usage can be of two types:

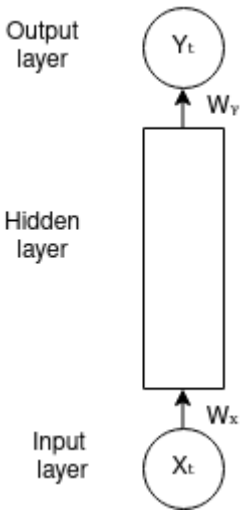
Attention is used between decoder steps: state h_{t-1} is used to compute attention and its output $c^{(t)}$, and both h_{t-1} and $c^{(t)}$ are passed to the decoder at step t.

Attention is used after RNN decoder step t before making a prediction. State h_t used to compute attention and its output $c^{(t)}$. Then h_t is combined with $c^{(t)}$ to get an updated representation $\tilde{h}_t$, which is used to get a prediction.

RNN

Recurrent Neural Networks (RNNs) were introduced to address the limitations of traditional neural networks, such as FeedForward Neural Networks (FNNs), when it comes to processing sequential data. FNN takes inputs and process each input independently through a number of hidden layers without considering the order and context of other inputs. Due to which it is unable to handle sequential data effectively and capture the dependencies between inputs. As a result, FNNs are not well-suited for sequential processing tasks such as, language modeling, machine translation, speech recognition, time series analysis, and

many other applications that requires sequential processing. To address the limitations posed by traditional neural networks, RNN comes into the picture.
RNN overcome these limitations by introducing a recurrent connection that allow information to flow from one time-step to the next. This recurrent connection enables RNNs to maintain internal memory, where the output of each step is fed back as an input to the next step, allowing the network to capture the information from previous steps and utilize it in the current step, enabling model to learn temporal dependencies and handle input of variable length.



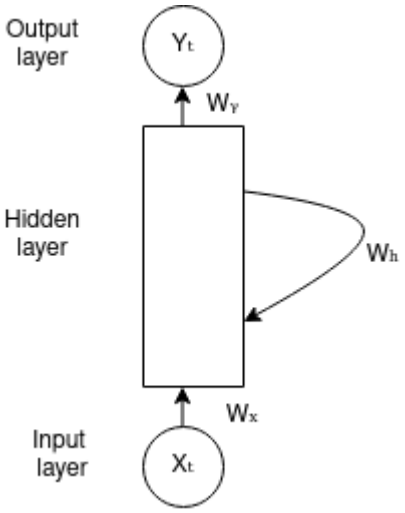


fig 1: FeedForward Neural Network (FNN). Image by Author

fig 2: Recurrent Neural Network (RNN). Image by Author
Architecture Of RNN
For more clear understanding of the concept of RNN, let's look at the unfolded RNN diagram.

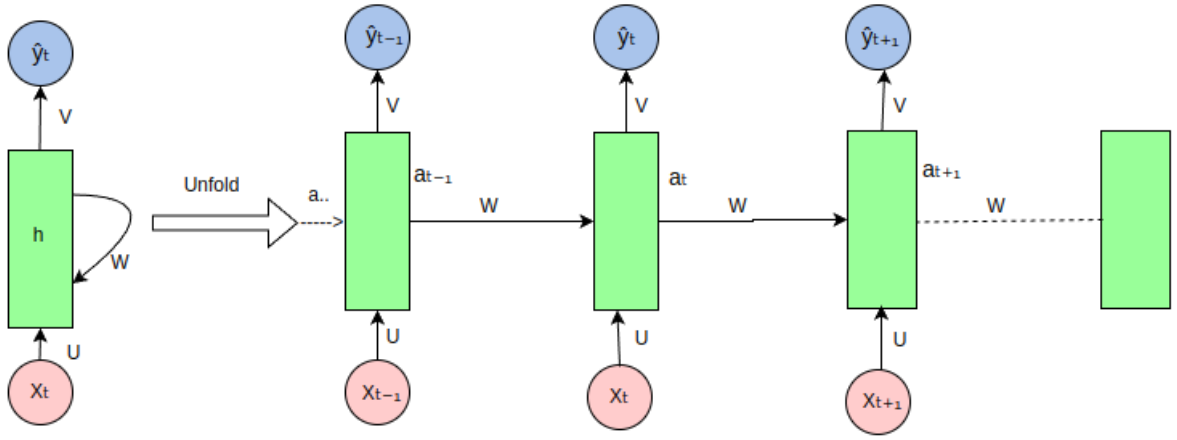


fig 3: RNN Unfolded. Image by Author.
The RNN takes an input vector X and the network generates an output vector y by scanning the data sequentially from left to right, with each time step updating the hidden state and producing an output. It shares the same parameters across all time steps. This means that, the same set of parameters, represented by U, V, W is used consistently throughout the network. U represents the weight parameter governing the connection from input layer X to the hidden layer h , W represents the weight associated with the connection between hidden layers, and V for the connection from hidden layer h to output layer y. This sharing of

parameters allows the RNN to effectively capture temporal dependencies and process sequential data more efficiently by retaining the information from previous input in its current hidden state.

At each time step t, the hidden state a_t is computed based on the current input x_t , previous hidden state a_{t-1} and model parameters as illustrated by the following formula:

$a_t = f(a_{t-1}, x_t; \theta)$ — — — — — (1)

It can also be written as,

$a_t = f(U * X_t + W* a_{t-1} + b)$

where,

- a_t represents the output generated from the hidden layer at time step t .
- x_t is the input at time step t.
- θ represents a set of learnable parameters(weights and biases).
- U is the weight matrix governing the connections from the input to the hidden layer; $U \in \theta$
- W is the weight matrix governing the connections from the hidden layer to itself (recurrent connections); $W \in \theta$
- V represents the weight associated with connection between hidden layer and output layer; $V \in \theta$
- a_{t-1} is the output from hidden layer at time t-1.
- b is the bias vector for the hidden layer; $b \in \theta$
- f is the activation function.

For a finite number of time steps T=4, we can expand the computation graph of a Recurrent Neural Network, illustrated in Figure 3, by applying the equation (1) T-1 times.

$a_4 = f(a_3, x_4; \theta)$ — — — — — (2)

Equation (2) can be expanded as,

$a_4 = f(U * X_4 + W* a_3 + b)$

$a_3 = f(U * X_3 + W* a_2 + b)$

$a_2 = f(U * X_2 + W* a_1 + b)$

The output at each time step t, denoted as y_t is computed based on the hidden state output a_t using the following formula,

$\hat{y}_t = f(a_t; \theta)$ — — — — — (3)

Equation (3) can be written as,

$\hat{y}_t = f(V * a_t + c)$

when t=4, $\hat{y}_4 = f(V * a_4 + c)$

where,

- $\hat{y}_t$ is the output predicted at time step t.
- V is the weight matrix governing the connections from the hidden layer to the output layer.
- c is the bias vector for the output layer.

Backpropagation Through Time (BPTT)

Backpropagation involves adjusting the model's parameters (weights and biases) based on the error between predicted output and the actual target value. The goal of backpropagation is to improve the model's performance by minimize the loss function. Backpropagation Through Time is a special variant of backpropagation used to train RNNs, where the error is propagated backward through time until the initial time step t=1. Backpropagation involves two key steps: forward pass and backward pass.

1. Forward Pass: During forward pass, the RNN processes the input sequence through time, from t=1 to t=n, where n is the length of input sequence. In each forward propagation, the following calculation takes place

$a_t = U * X_t + W * a_{t-1} + b$

$a_t = \tanh(a_t)$

$\hat{y}_t = \text{softmax}(V * a_t + c)$

After processing the entire sequence, RNN generates a sequence of predicted outputs $\hat{y} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_t]$. Loss is then computed by comparing predicted output $\hat{y}$ at each time step with actual target output y. Loss function given by,

$L(y, \hat{y}) = (1/t) * \Sigma(y_t - \hat{y}_t)^2$ — — —— — -> MSE Loss

2. Backward Pass: The backward pass in BPTT involves computing the gradients of the loss function with respect to the network's parameters (U, W, V and biases) over each time step. Let's explore the concept of backpropagation through time by computing the gradients of loss at time step t=4. The figure below also serves as an illustration of backpropagation for time step 4.

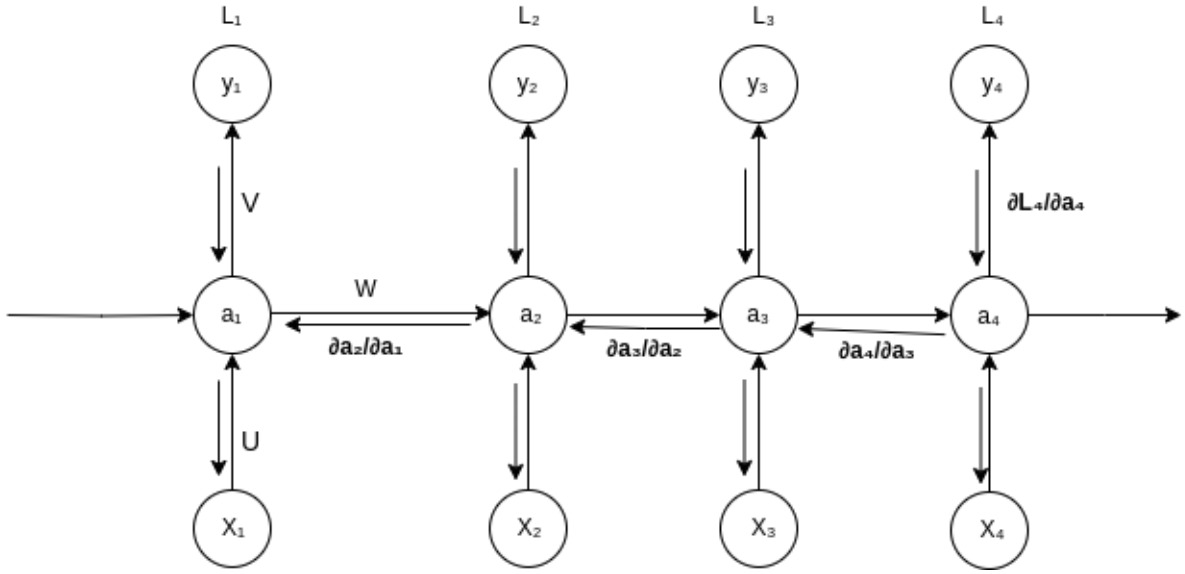


fig 4: Back Propagation Through Time (BPTT). Image by Author

Derivative of loss L w.r.t V

Loss L is a function of predicted value $\hat{y}$, so using the chain rule $\partial L/\partial V$ can be written as,

$$\partial L/\partial V = (\partial L/\partial \hat{y}) * (\partial \hat{y}/\partial V)$$

Derivative of loss L w.r.t W

Applying the chain rule of derivatives $\partial L/\partial W$ can be written as follows: The loss at the 4th time step is dependent upon $\hat{y}$ due to the fact that the loss is calculated as a function of $\hat{y}$, which is in turn dependent on the current time step's hidden state a_4, a_4 is influenced by both W and a_3, and again a_3 is connected to both a_2 and W,and a_2 depends on a_1 and also on W.

$$\partial L_4/\partial W = (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial a_1 * \partial a_1/\partial W)$$

Derivative of loss L w.r.t U

Similarly, $\partial L/\partial U$ can be written as,

$$\partial L_4/\partial U = (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial a_1 * \partial a_1/\partial U)$$

Here we're summing up the gradients of loss across all time steps which represents the key difference between BPTT and regular backpropagation approach.

Limitations of RNN

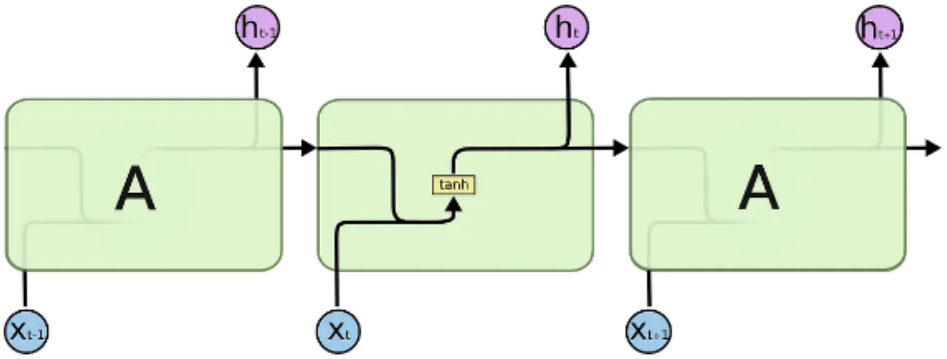
During backpropagation, gradients can become too small, leading to the vanishing gradient problem, or too large, resulting in the exploding gradient problem as they propagate backward through time. In the case of vanishing gradients, the issue is that the gradient may become too small where the network struggles to capture long-term dependencies effectively. It can still converge during training but it may take a very very long time. In

contrast, in exploding gradient problem, large gradient can lead to numerical instability during training, causing the model to deviate from the optimal solution and making it difficult for the network to converge to global minima.

No of parameters in RNN
input_vector is of m dimensions and there is n number of hidden units (RNN units in RNN cell) also called as number of units
input_vector dimensions x number of hidden units = m x n
bias unit = number of hidden units
hidden units x hidden units (this is the recurrent connections — the weights of the previous timestep (or) previous sequence are carried out along with the next timestep's input — but don't assume that each and every timestep will have different weight matrices — This is the recurrent network, the same network will be recurrently used until the end of the sequence.
Total parameters = input_vector x number of units + bias (number of units) + number of units x number of units (recurrent connections)
Total parameters = (m x n) + n + (n x n) = mn + n + n**2 = n(m+1+n) = n(m+n+1)
Example: 1 — As per the above example in the above image, there are 4 hidden units — number of units(n) = 4, and the input vector — m = 3.
So, the total parameters = 12+4+16 = 32
Example: 2 — If the input vector to RNN is of 3 dimensions and the number of units is 2,
The total parameters = 6+2+4 = 12

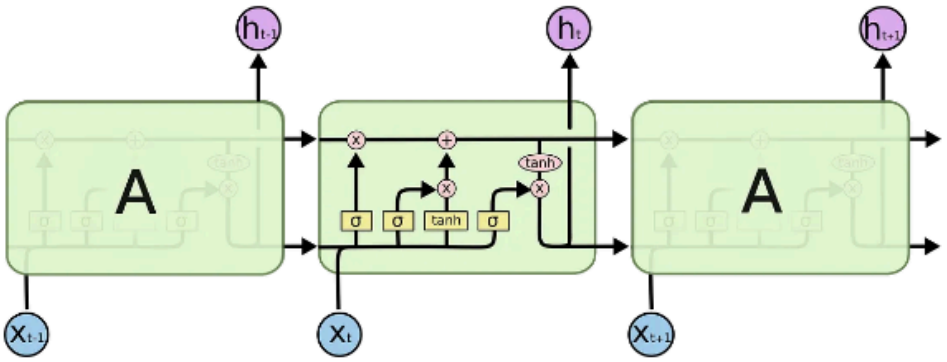
LSTM

LSTMs come to the rescue to solve the vanishing gradient problem. It does so by ignoring (forgetting) useless data/information in the network. The LSTM will forget the data if there is no useful information from other inputs (prior sentence words). When new information comes, the network determines which information to be overlooked and which to be remembered.

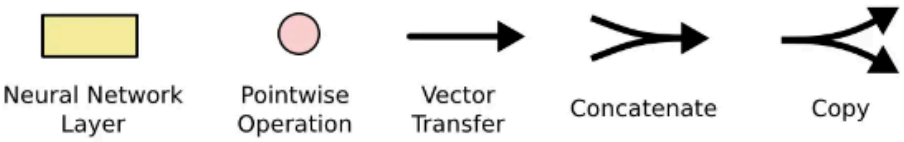


RNN network; Source: colah's blog

In LSTMs, instead of just a simple network with a single activation function, we have multiple components, giving power to the network to forget and remember information.



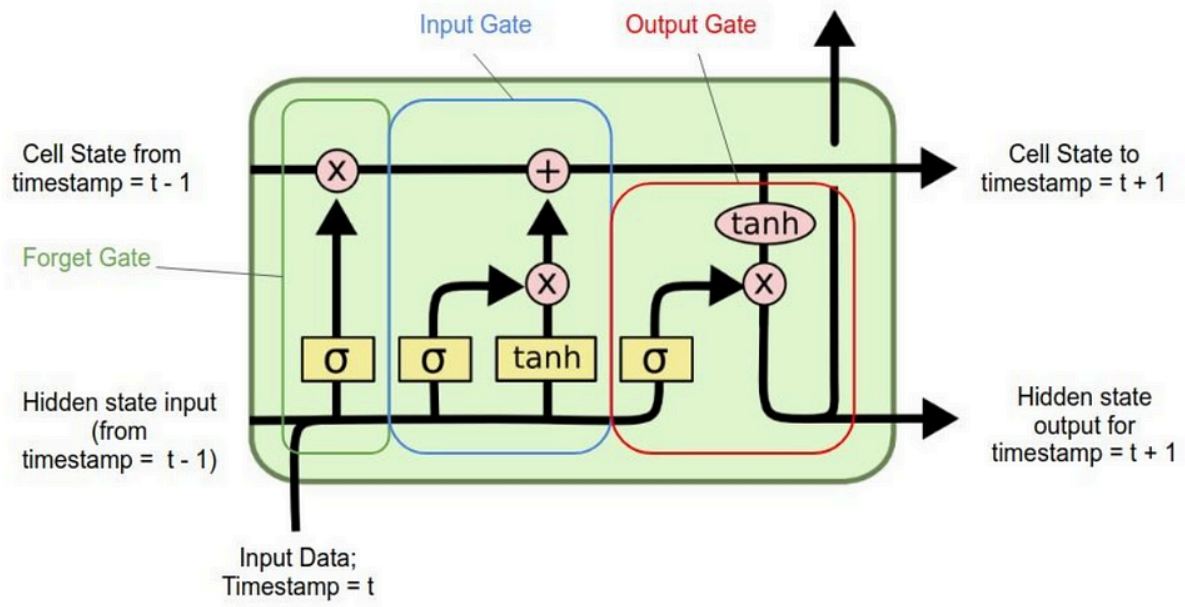
LSM network; Source: colah's blog



Notations used

LSTMs have 4 different components, namely

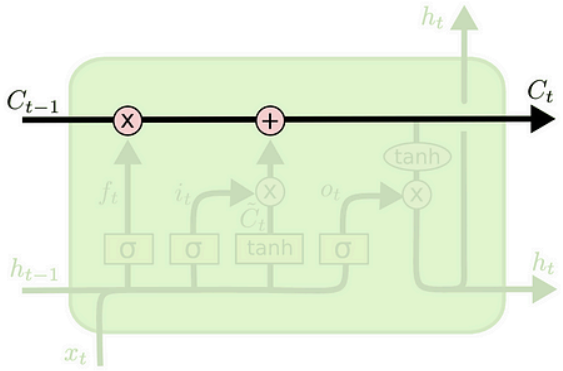
1. Cell state (Memory cell)
2. Forget gate
3. Input gate
4. Output gate



Let's understand these components, one by one.

1. Cell State (Memory cell)

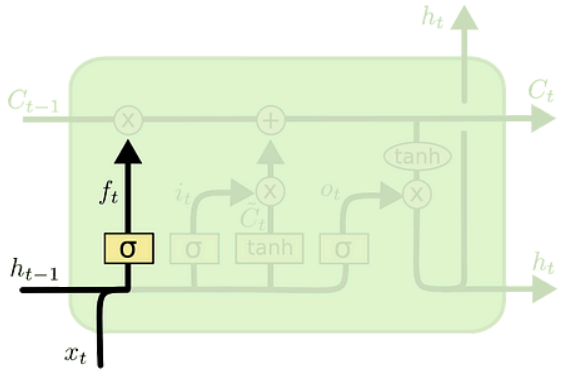
It is the first component of LSTM which runs through the entire LSTM unit. It kind of can be thought of as a conveyer belt.



This cell state is responsible for remembering and forgetting. This is based on the context of the input. This means that some of the previous information should be remembered while some of them should be forgotten and some of the new information should be added to the memory. The first operation (X) is the pointwise operation which is nothing but multiplying the cell state by an array of [-1, 0, 1]. The information multiplied by 0 will be forgotten by the LSTM. Another operation is (+) which is responsible to add some new information to the state.

2. Forget Gate

The forget LSTM gate, as the name suggests, decides what information should be forgotten. A sigmoid layer is used to make this decision. This sigmoid layer is called the "forget gate layer".

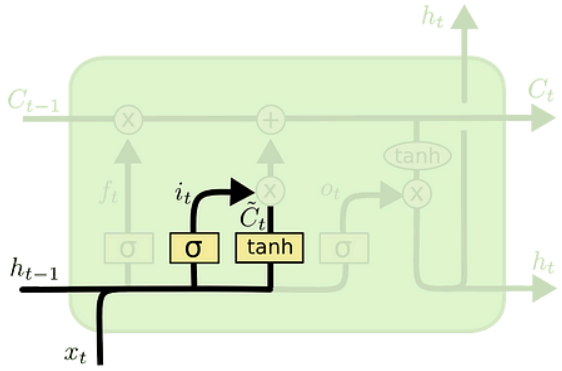


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

It does a dot product of h(t-1) and x(t) and with the help of the sigmoid layer, outputs a number between 0 and 1 for each number in the cell state C(t-1). If the output is a '1', it means we will keep it. A '0' means to forget it completely.

3. Input gate

The input gate gives new information to the LSTM and decides if that new information is going to be stored in the cell state.



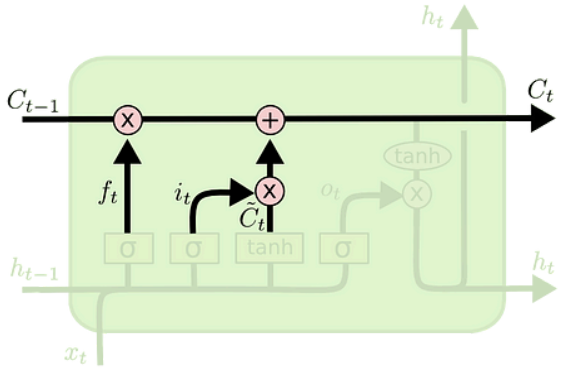
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

This has 3 parts-

1. A sigmoid layer decides the values to be updated. This layer is called the "input gate layer"
2. A tanh activation function layer creates a vector of new candidate values, Č(t), that could be added to the state.
3. Then we combine these 2 outputs, i(t) * Č(t), and update the cell state.

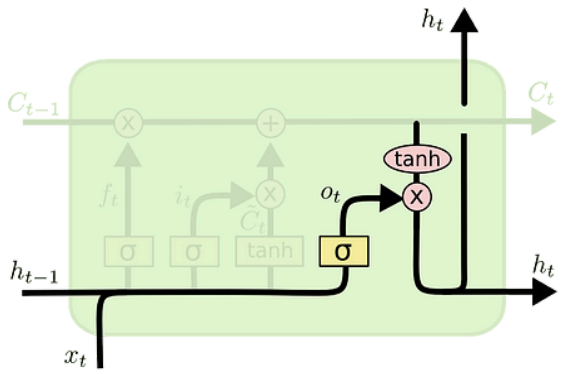
The new cell state C(t) is obtained by adding the output from forget and input gates.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

4. Output gate

The output of the LSTM unit depends on the new cell state.

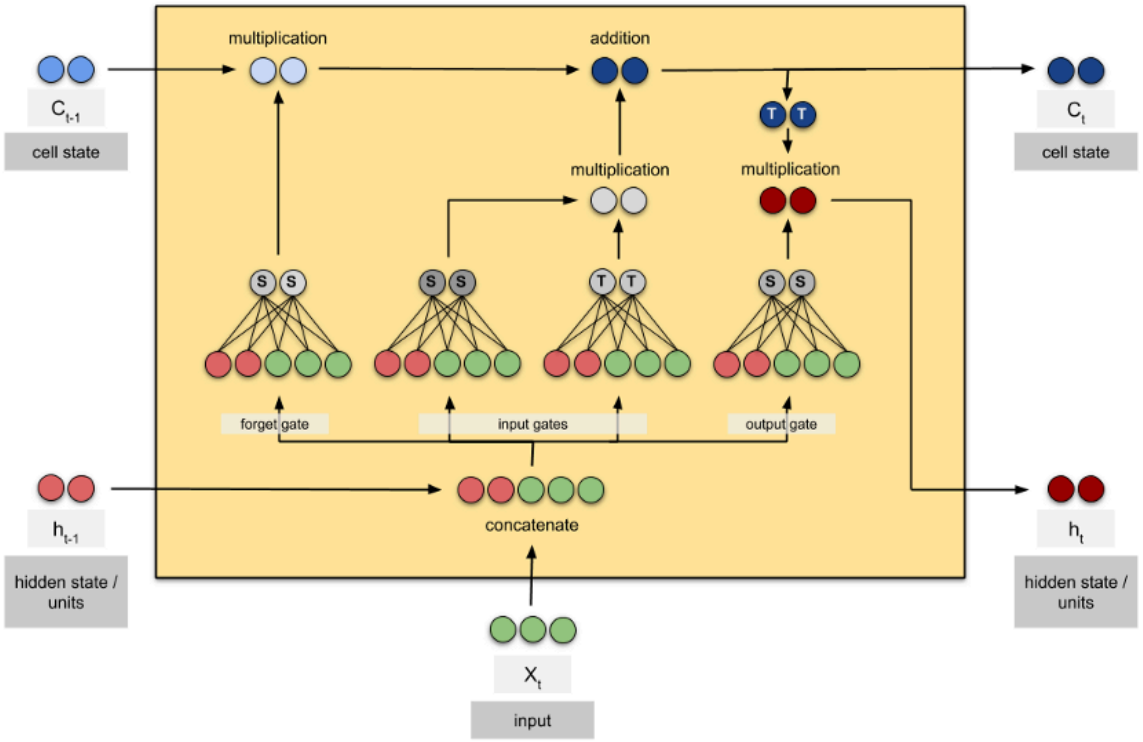


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

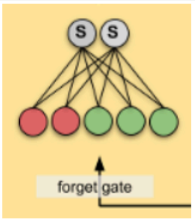
$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM Output gate; Source: [colah's blog]

First, a sigmoid layer decides what parts of the cell state we're going to output. Then, a tanh layer is used on the cell state to squash the values between -1 and 1, which is finally multiplied by the sigmoid gate output.



No of parameters in LSTM



- As seen in above figure,
  - there are Hidden state values (**2** h_{t-1} red circles) and
  - Input values (**3** x_t green circles)
- Total 5 numbers (**2** h_{t-1} + **3** x_t) are inputted to **a dense layer**
- Output layer has **2 values** (which **must be equal** to the dimension of h_{t-1} Hidden state vector in the LSTM Cell)
- We can calculate the number of parameters in this dense layer as we did before:

  = $(h_{t-1} + x_t) \times h_{t-1} + h_{t-1}$

  = (2 + 3) × 2 + 2

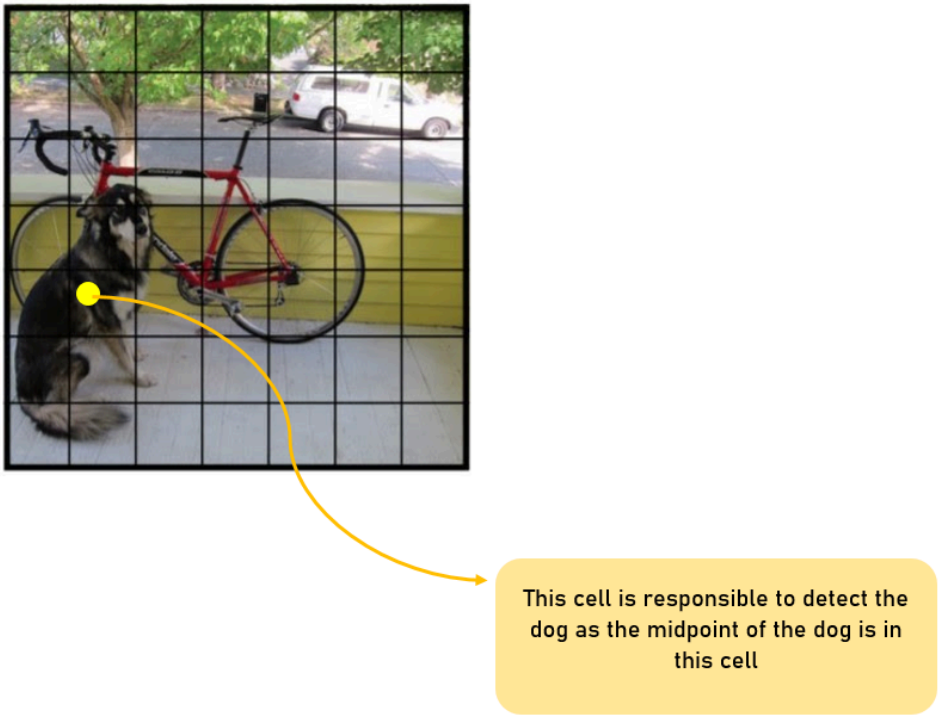
  = 12

- **Thus Forget Gate has 12 parameters (weights + biases)**

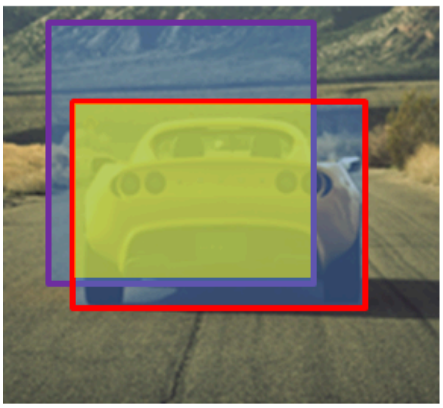
Since there are 4 gates in the LSTM unit which have exactly the same dense layer architecture, there will be
= 4 × 12
= 48 parameters

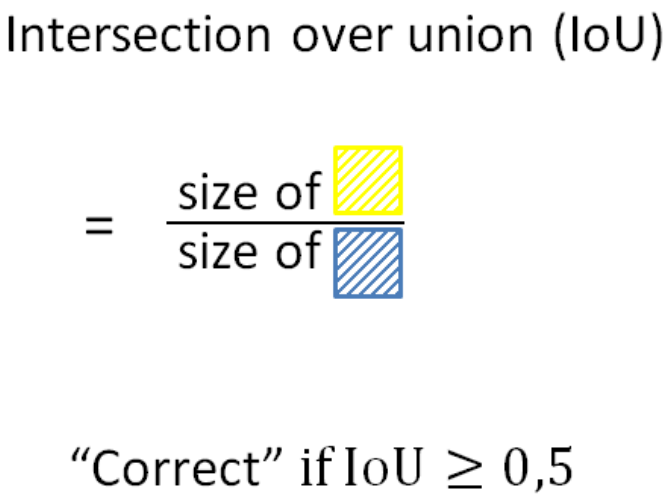
YOLOv3

Let us first understand how YOLO encodes its output,
1. Input image is divided into NxN grid cells. For each object present on image, one grid cell is responsible for predicting object.
2. Each grid predicts ‘B’ bounding box and ‘C’ class probabilities. And bounding box consist of 5 components (x,y,w,h,confidence)
(x,y) = coordinates representing center of box
(w,h) = width and height of box
Confidence = represents presence/absence of any object
Let us see these with an example,



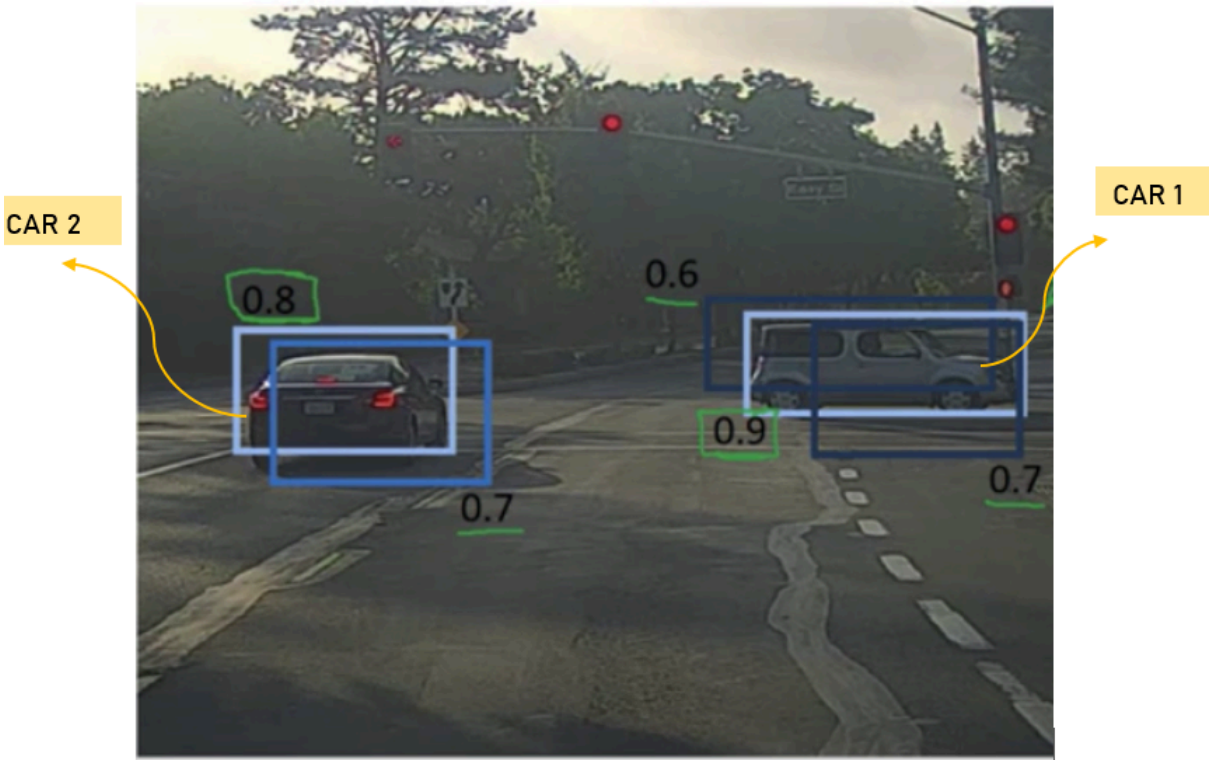
The cell which has center of object that cell determines or is responsible for detecting object.
Challenges in YOLO:
Question 1. How do we tell if the object detection algorithm is working well?
Solution: We have already seen this in previous article where we discussed R-CNN Family models, how do we evaluate object localization, it is by metric called Intersection over Union (IOU)
IOU = Area of Intersection / Area of Union





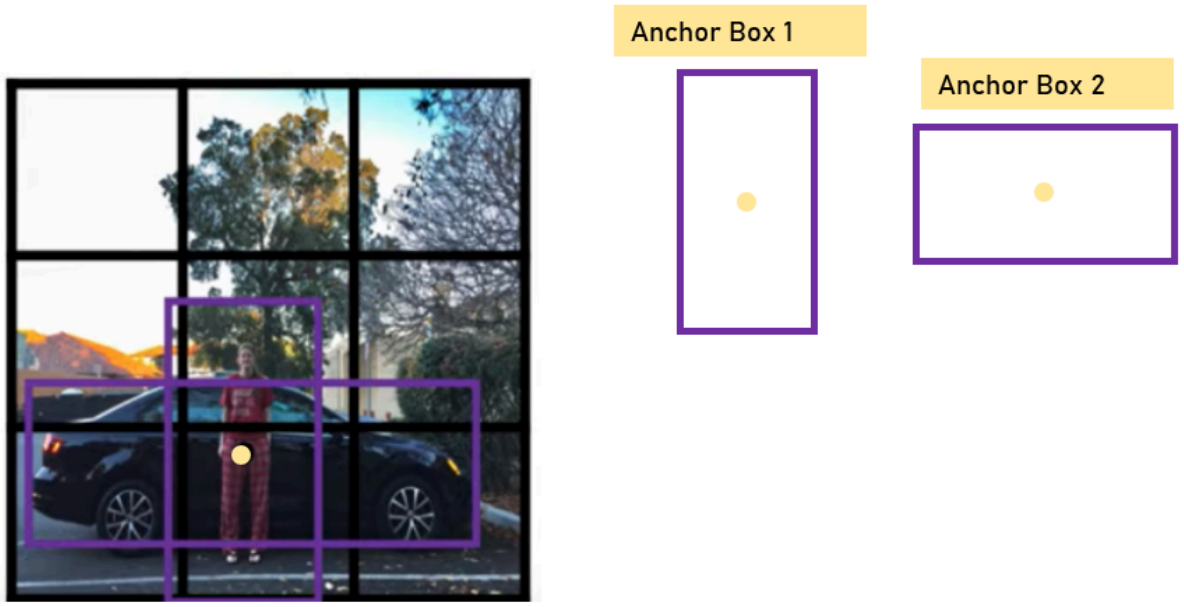
Question 2. There can be multiple bounding boxes for each object in an image, so how to deal with it?
Solution: We use Non-Max Suppression (NMS) which is the way to make sure that your algorithm detects your object only once.
We will see how this works,

Because, you are running image classification and localization algorithm on every grid cell, it is possible that many of the cells say that their ‘Pc’ Class Probability or chance of having object in that cell is highest.
So when we run algorithm, we might end up with multiple detections for same object.
So, what NMS does is that it cleans up other unwanted detections so we end up with one detection for particular object.
How does this NMS work?
1. First it looks for probabilities (Pc) associated with each of these detection for particular object
2. It takes largest ‘Pc’ which is most confident detection for the object
3. Having done that, the NMS part looks for all remaining bounding boxes and chooses all those bounding boxes which has high Intersection over Union (IOU) with the bounding box of highest ‘Pc’ and suppresses them.
4. Then we look for remaining bounding box and find highest ‘Pc’ and again NMS looks for remaining bounding boxes which has high IOU with bounding box of high ‘Pc’ and then they will get suppressed.
By doing this for every object we get only one bounding box for each object.

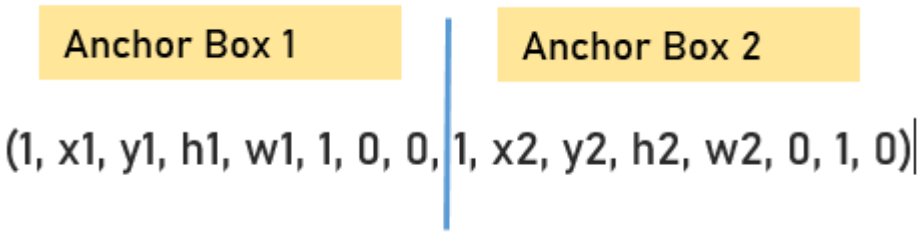


So for this example:
1. It takes largest Pc which is 0.9 in this case
2. It check IOU for all the remaining bounding boxes (i.e. for 0.6, 0.7 for Car 1 and 0.8, 0.7 for Car 2)
3. Now, NMS will suppress 0.6 and 0.7 for car 1 as they have high IOU with respect to bounding box of Pc=0.9, so like this we get only one bounding box for car 1 which is highlighted in the image.
4. Next, for remaining bounding boxes we have highest Pc=0.8 for car2 and again we check IOU for remaining boxes (i.e. 0.9 for car1 and 0.7 for car2)

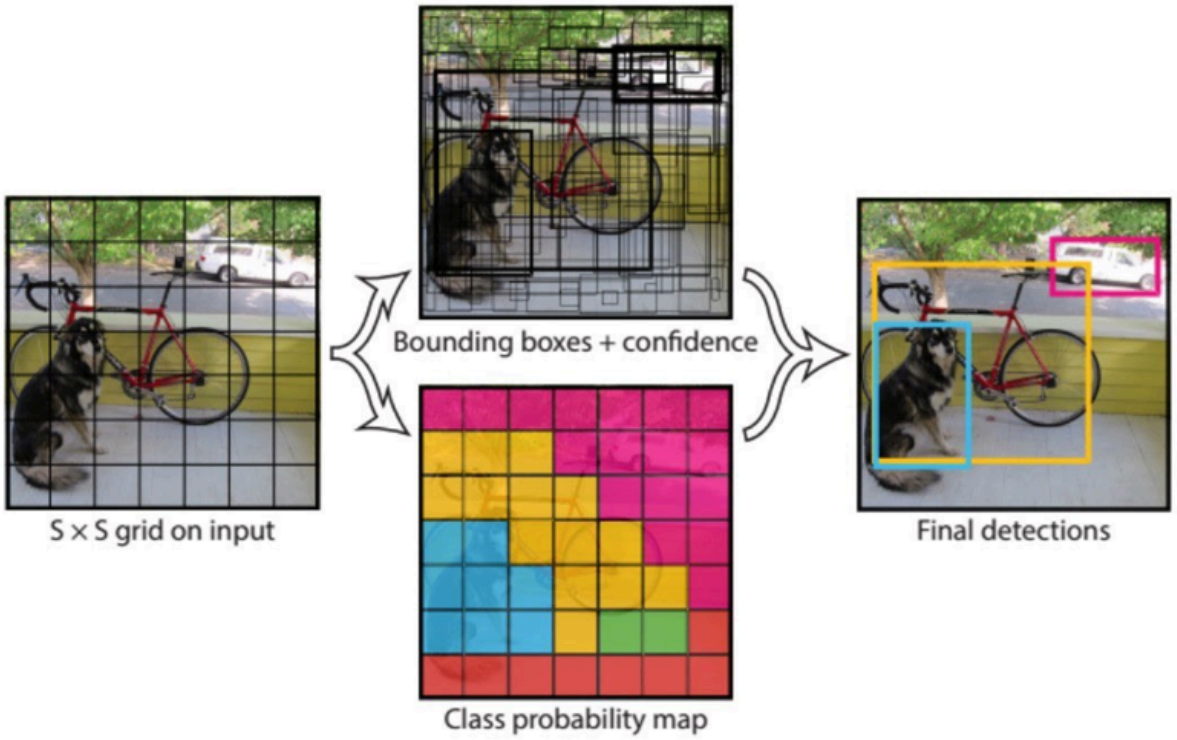
5. Now, NMS will suppress 0.7 as it has high IOU with respect to bounding box of Pc=0.8. And we get only one bounding box for car 2 as well.
Question 3. What if we have multiple objects in a single cell? i.e. what if we have overlapping objects and midpoint of both objects lie in single grid cell?
Solution: We use multiple anchor boxes for solving this,
Each cell represents this output (Pc, x, y, h, w, c1, c2, c3) which is a vector of shape (8, 1) i.e. 8 rows and 1 column. c1, c2, c3 are the different classes say person, car, bike.
So, the shape of bounding box can change depending on number of classes.



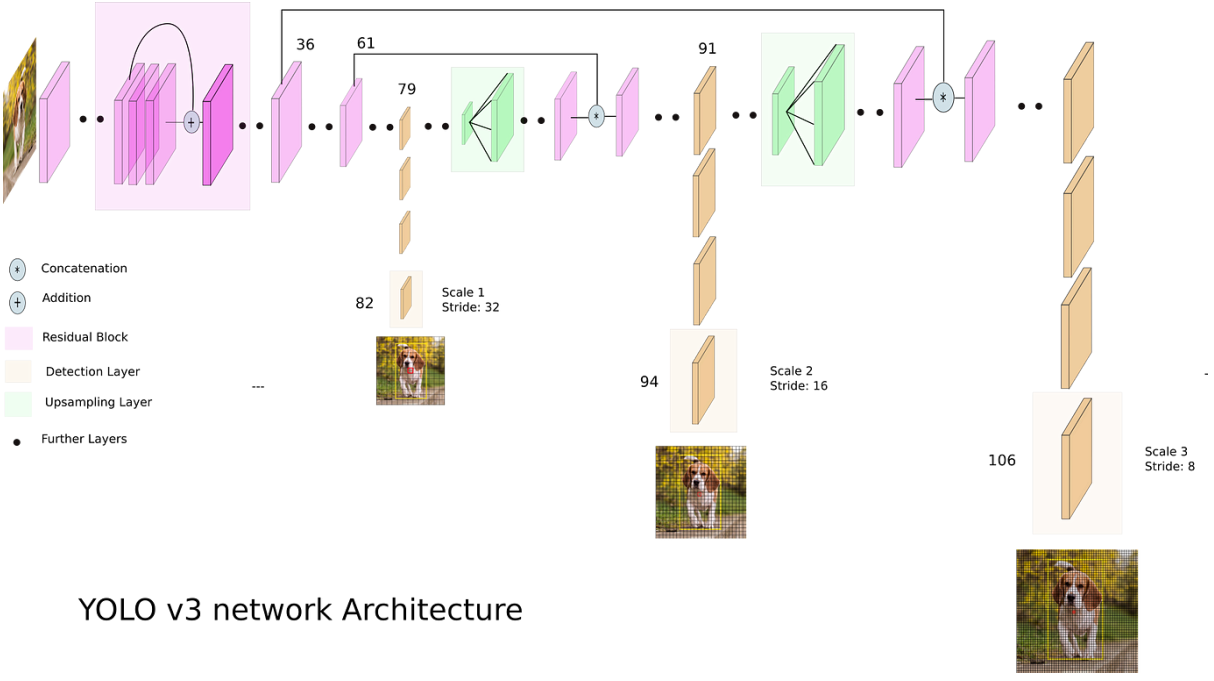
Now, each cell won't be able to output 2 detection so have to pick any one of the two detections to output.
With the idea of anchor boxes what you are going to do is predefine 2 different shapes called Anchor Box 1 and Anchor Box 2. By this we can do two predictions with 2 anchor boxes.
In general we can use more anchor boxes, to capture variety of shapes the object has.
So, for two anchor boxes how our output will be in case of 3 classes,
The output will be vector of size (16, 1) and the vector contains, (Pc1, x1, y1, h1, w1, c1, c2, c3, Pc2, x2, y2, h2, w2, c1, c2, c3)
Let's say c1=person, c2=car, c3=bike
In our example as we have person and car so the output will be,



Combining all these steps we get our YOLO algorithm:



Limitations of YOLO:
YOLO can only predict a limited number of bounding boxes per grid cell, 2 in the original research paper. And though that number can be increased, only one class prediction can be made per cell, limiting the detections when multiple objects appear in a single grid cell. Thus, it struggles with bounding groups of small objects, such as flocks of birds, or multiple small objects of different classes.
Architecture of YOLOv3:

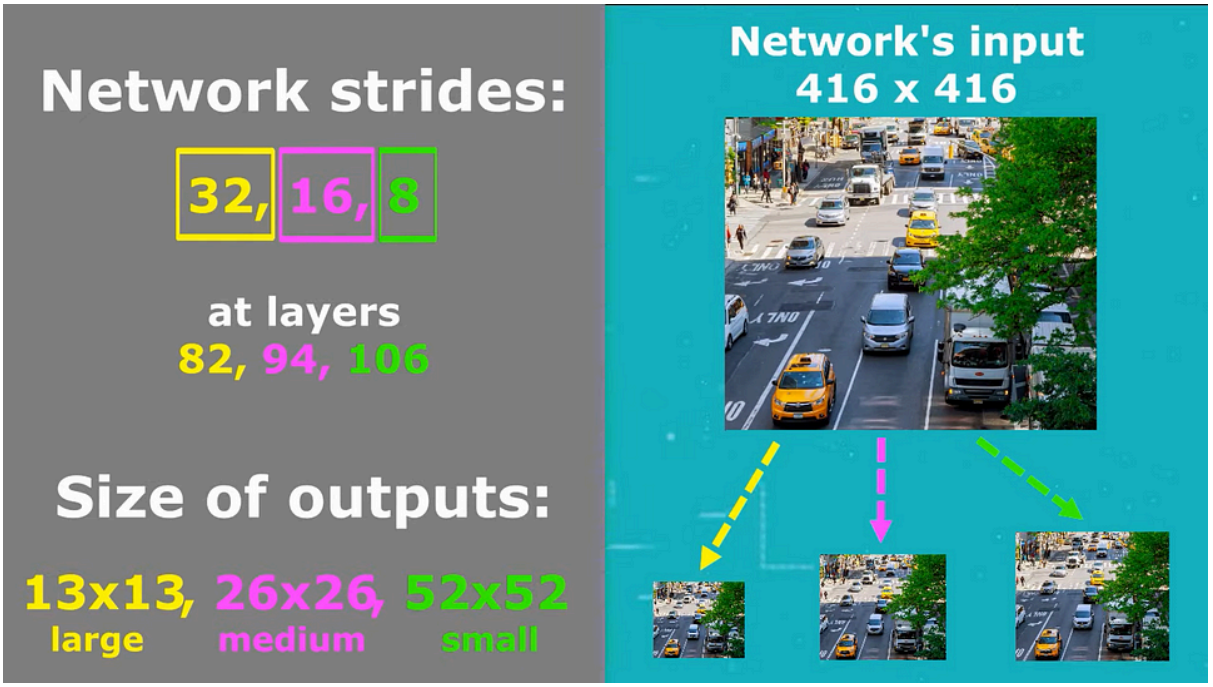


YOLO v3 network Architecture

YOLO v3 uses a variant of Darknet, which originally has 53 layer network trained on ImageNet. For the task of detection, 53 more layers are stacked onto it, giving us a 106 layer fully convolutional underlying architecture for YOLO v3.
The detections are made at three layers 82nd, 94th and 106th layer.
Convolution layers in YOLOv3

- It contains 53 convolutional layers, each followed by batch normalization layer and Leaky ReLU activation.
- Convolution layer is used to convolve multiple filters on the images and produces multiple feature maps
- No form of pooling is used and a convolutional layer with stride 2 is used to downsample the feature maps.
- It helps in preventing loss of low-level features often attributed to pooling.

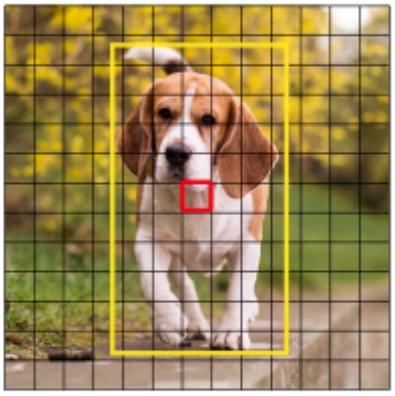
Now let's look at the input, how does it look like,
The input is batch of images of shape (n, 416, 416, 3) where,
n=number of images, (416,416) = (width, height) and 3 channels (RGB).
The width and height can be changed to any number which is divisible by 32. These numbers (width, height) are also called as input network size.
Why divisible by 32?
We will see this in a moment.
Increase in resolution of input might increase accuracy after training. Input images can be of any size, we don't need to resize them before feeding them to the network, they all will be resized according to input network size.
How the network detects objects?
YOLOv3 makes detections at 3 different places in the network. These are layers 82nd, 94th and 106th layer. Network down samples input image by following factors 32, 16 and 8 at 82nd, 94th, 106th layer accordingly, these numbers are called strides to the network and they show how the output at 3 places in the network are smaller than network input.



For network input (416, 416),
For 82nd layer the stride is 32 and the output size is 13x13 and it is responsible to detect large objects

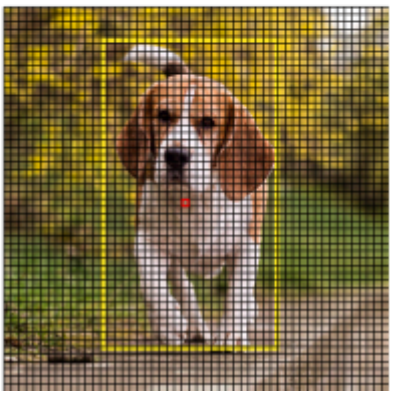
For 94th layer the stride is 16 and the output size is 26x26 and it is responsible to detect medium objects
For 106th layer the stride is 8 and the output size is 52x52 and it is responsible to detect small objects
This is the reason why the network input must be divisible by 32, because if it is divisible by 32 then it is also divisible by 16 and 8 as well.

Prediction Feature Maps at different Scales



13 x 13

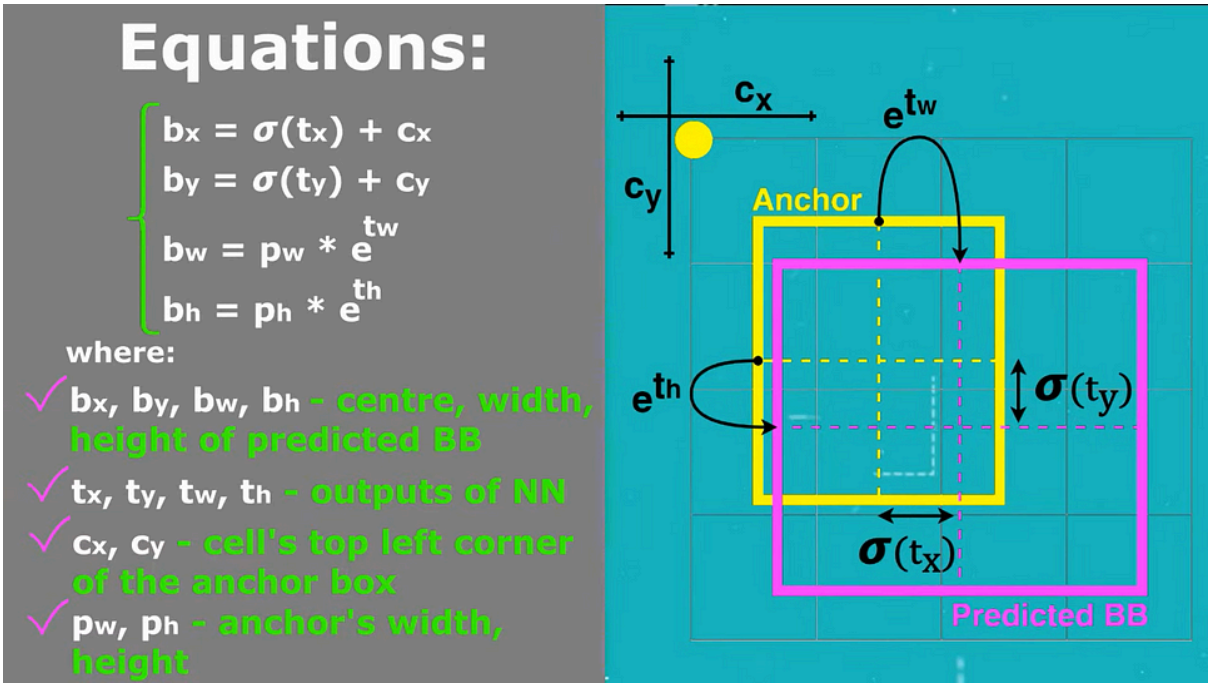
26 x 26



52 x 52

Here we can see, why 13x13, 26x26, 52x52 detect large, medium and small objects
Now let us see, what detection kernels are,
So to produce outputs YOLOv3 applies 1x1 kernels(filters) at three output layers in the network. 1x1 kernels are applied to down sampled images so that our output has same spatial dimensions 13x13, 26x26 and 52x52

The shape of the detection kernels also has its depth that is calculated by following formula, (b*(5+c)) where, b = Number of Bounding Boxes, c = number of classes, 80 (for COCO dataset)
Each bounding box has (5+c attributes)
For YOLOv3 it predicts 3 bounding boxes for every cell of each of these 3 feature maps (i.e. for 13x13, 26x26, 52x52 feature maps)
As, b = 3, c = 80, we get (3*(5+80)) = 255 attributes.
Now, we can say each feature maps produced by detection kernels at 3 separate places (output layers) in network has one more dimension depth that incorporates 255 attributes of bounding boxes for COCO dataset and shapes of these feature maps are as following, (13x13x255), (26x26x255), (52x52x255).
Let us now see, Grid Cells, simply, they are detection cells.
We already know that YOLOv3 predicts 3 bounding box for every cell of feature maps. So what is task of YOLOv3 is identify the cell which contains center of the object.
Training YOLOv3,
When it is training it has one ground truth bounding box that is responsible for detecting for one object. So, we need to know which cells these bounding box belongs to.
For this YOLOv3, makes predictions at 3 scales, for strides 32, 16 and 8.
So, the cell which has center of the object is responsible for detecting that object.
Anchor Boxes used to predict bounding boxes, YOLOv3 uses predefined bounding boxes called as anchors/priors and also these anchors/priors are used to calculate real width and real height for predicted bounding boxes.
In total 9 anchor boxes are used, 3 anchor boxes for each scale, three biggest anchors for the first scale, the next three for the second scale, and the last three for the third. This means at each output layer every grid scale of feature map can predict 3 bounding boxes using 3 anchor boxes.
To calculate these anchors K-Means Clustering is applied in YOLOv3.
The width and height of anchors,
For, Scale 1: (116x90), (156x198), (373x326)
Scale 2: (30x61), (62x45), (59x119)
Scale 3: (10x13), (16x30), (33x23)
So, for, Scale 1: we have, 13x13x3 = 507 bounding box
Scale 2: we have, 26x26x3 = 2028 bounding box
Scale 3: we have, 52x52x3 = 8112 bounding box
In total, YOLOv3 predicts 10,847 boxes.
To predict real height and width of bounding box YOLOv3 calculates offsets also called as log-space transform. Let’s see that now,



So, to predict center coordinates of bounding boxes(bx, by) YOLOv3 passes outputs(tx, ty) through sigmoid function.
So, based on the above equations given in figure we get center coordinates and width and height of the bounding boxes.
And all the redundant bounding boxes from 10,847 boxes are suppressed using NMS.

Anchor Box Generation using K-Means

Anchor boxes are a set of predefined bounding boxes of a certain height and width. These boxes are defined to capture the scale and aspect ratio of specific object classes you want to detect and are typically chosen based on object sizes in your training datasets. During detection, the predefined anchor boxes are tiled across the image. The network predicts the probability and other attributes, such as background, intersection over union (IoU) and offsets for every tiled anchor box. The predictions are used to refine each individual anchor box. You can define several anchor boxes, each for a different object size. Anchor boxes are fixed initial boundary box guesses.
Steps of generating Anchor boxes

1. Prepare the bounding boxes
2. Define distance calculation function
3. Implement K-Means alghorithm
4. Train and Evaluate the results

Step 1. Prepare the bounding boxes
We cannot rely on the original bounding boxes cuz image sizes are different. At first, should define a specific image size for all images, it might be (512, 512) or (640, 640) according to the input of the network. Resize the image by keeping the aspect ratio and additionally pad zeros to these areas where no pixel values fit the specified size. Finally redefine bounding boxes according to the resized, padded image.
Setp 2. Define distance calculation function

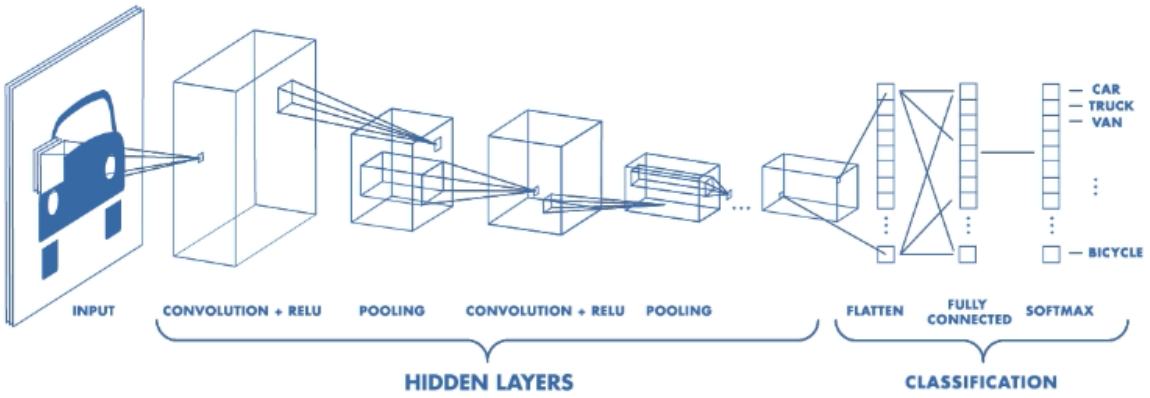
As a distance calculation between bounding boxes use IOU(Intersection over Union)
Step 3. Implement K-Means alghorithm
Principle of k-means:
The K-means algorithm is a typical distance-based clustering algorithm. The distance is used as the evaluation index of similarity. The closer the distance between two objects is, the greater the similarity. The algorithm considers clusters to be composed of objects that are close together, thus making compact and independent clusters the ultimate goal.
Step 4. Train and Evaluate the results
Evaluate by using elbow method to choose optimal number of anchor boxes. Ideally if IoU more than 60% then you can specify the number of anchors.

CNN

A Convolutional Neural Network, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be.

The human brain processes a huge amount of information the second we see an image. Each neuron works in its own receptive field and is connected to other neurons in a way that they cover the entire visual field. Just as each neuron responds to stimuli only in the restricted region of the visual field called the receptive field in the biological vision system, each neuron in a CNN processes data only in its receptive field as well. The layers are arranged in such a way so that they detect simpler patterns first (lines, curves, etc.) and more complex patterns (faces, objects, etc.) further along.

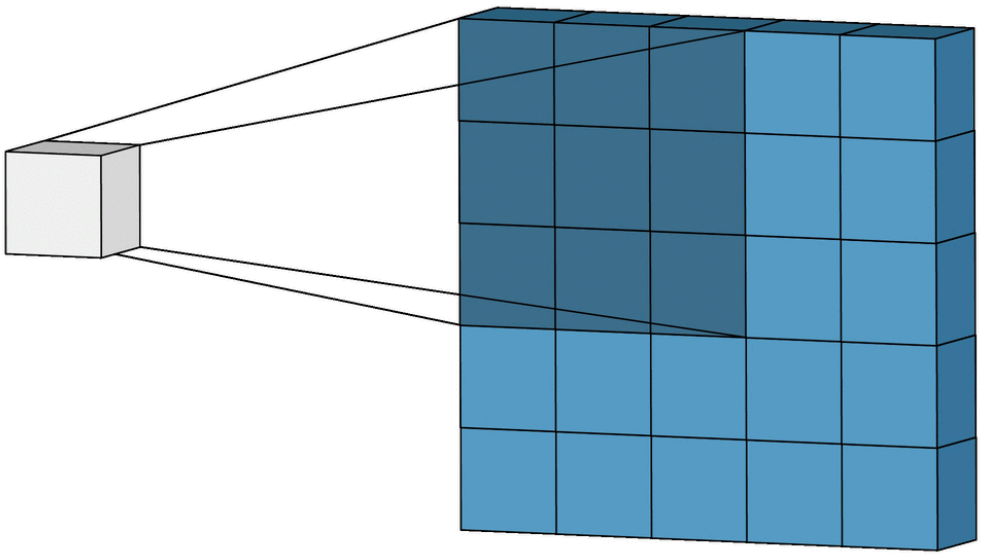
A CNN typically has three layers: a convolutional layer, a pooling layer, and a fully connected layer.



Convolution Layer

The convolution layer is the core building block of the CNN. It carries the main portion of the network's computational load.

This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field. The kernel is spatially smaller than an image but is more in-depth. This means that, if the image is composed of three (RGB) channels, the kernel height and width will be spatially small, but the depth extends up to all three channels.

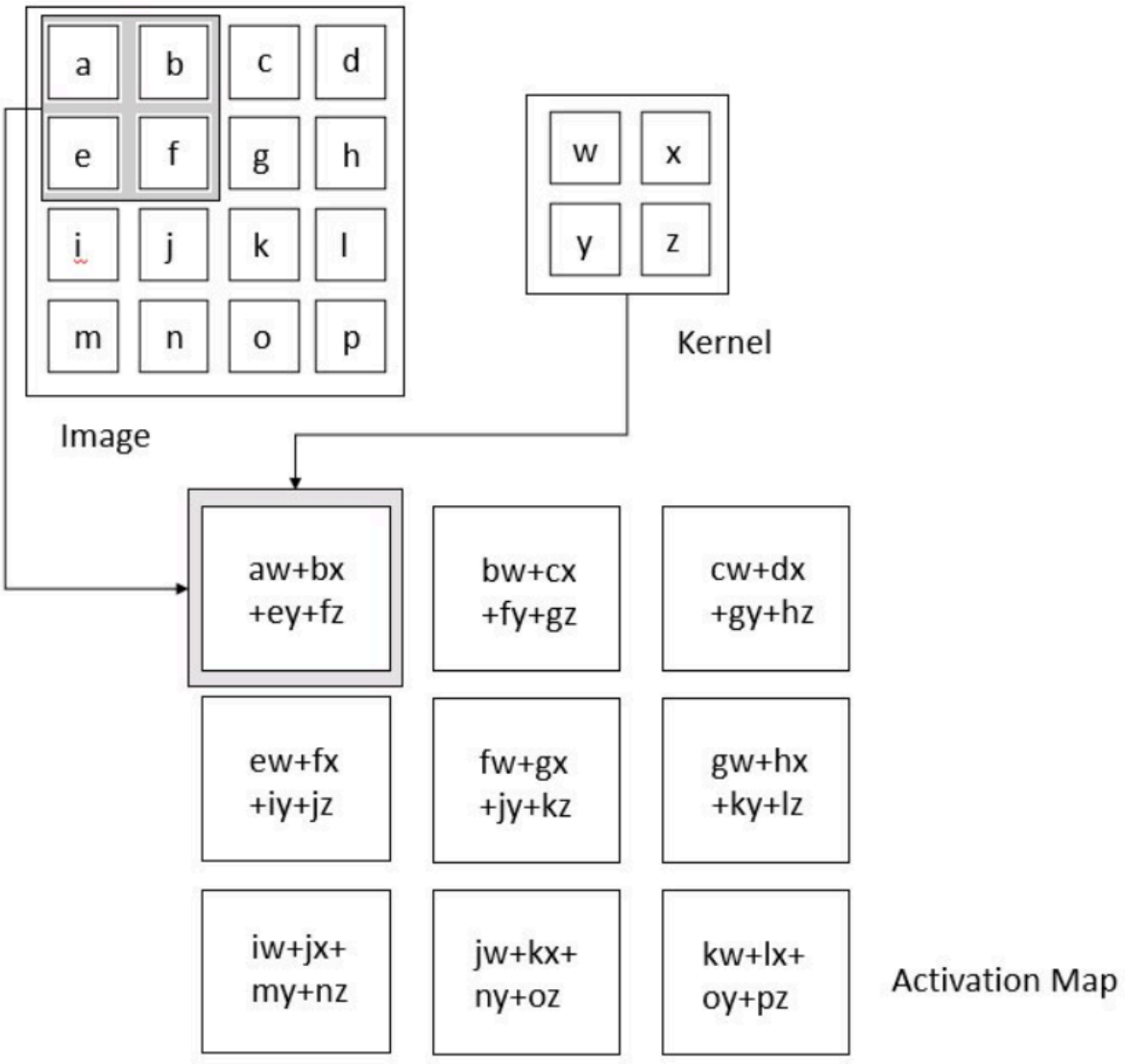


During the forward pass, the kernel slides across the height and width of the image-producing the image representation of that receptive region. This produces a

two-dimensional representation of the image known as an activation map that gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is called a stride.
If we have an input of size W x W x D and Dout number of kernels with a spatial size of F with stride S and amount of padding P, then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F + 2P}{S} + 1$$

Formula for Convolution Layer
This will yield an output volume of size Wout x Wout x Dout.



Motivation behind Convolution

Convolution leverages three important ideas that motivated computer vision researchers: sparse interaction, parameter sharing, and equivariant representation. Let's describe each one of them in detail.

Trivial neural network layers use matrix multiplication by a matrix of parameters describing the interaction between the input and output unit. This means that every output unit interacts with every input unit. However, convolution neural networks have sparse interaction. This is achieved by making kernel smaller than the input e.g., an image can have millions or thousands of pixels, but while processing it using kernel we can detect meaningful information that is of tens or hundreds of pixels. This means that we need to store fewer parameters that not only reduces the memory requirement of the model but also improves the statistical efficiency of the model.

If computing one feature at a spatial point (x1, y1) is useful then it should also be useful at some other spatial point say (x2, y2). It means that for a single two-dimensional slice i.e., for creating one activation map, neurons are constrained to use the same set of weights. In a traditional neural network, each element of the weight matrix is used once and then never revisited, while convolution network has shared parameters i.e., for getting output, weights applied to one input are the same as the weight applied elsewhere.

Due to parameter sharing, the layers of convolution neural network will have a property of equivariance to translation. It says that if we changed the input in a way, the output will also get changed in the same way.

Pooling Layer

The pooling layer replaces the output of the network at certain locations by deriving a summary statistic of the nearby outputs. This helps in reducing the spatial size of the representation, which decreases the required amount of computation and weights. The pooling operation is processed on every slice of the representation individually.

There are several pooling functions such as the average of the rectangular neighborhood, L2 norm of the rectangular neighborhood, and a weighted average based on the distance from the central pixel. However, the most popular process is max pooling, which reports the maximum output from the neighborhood.

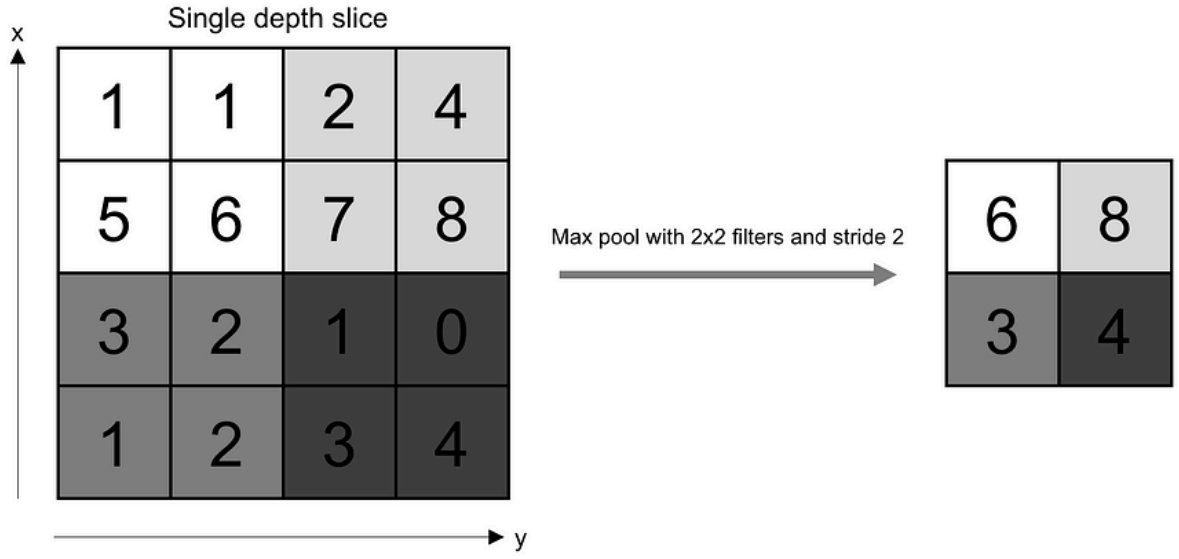


Figure 4: Pooling Operation (Source: O'Reilly Media)

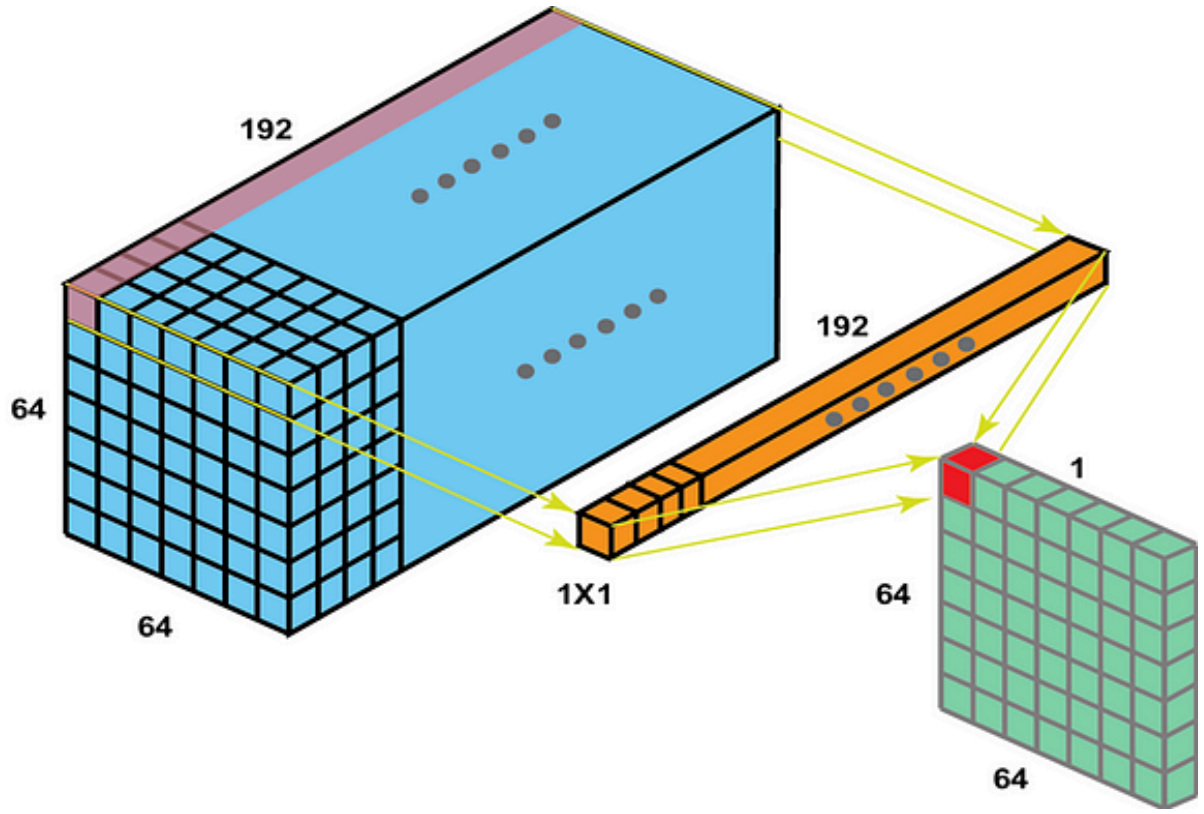
If we have an activation map of size W x W x D, a pooling kernel of spatial size F, and stride S, then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F}{S} + 1$$

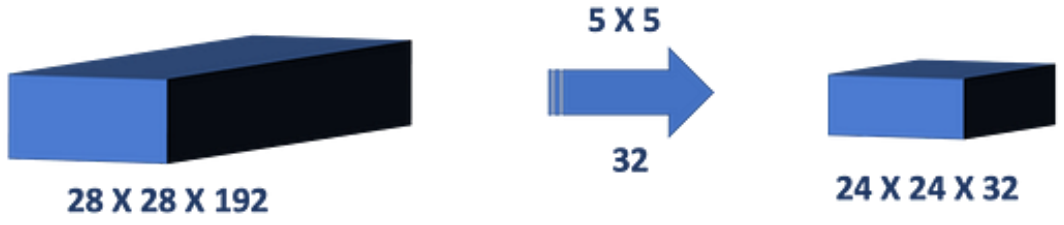
Formula for Padding Layer
This will yield an output volume of size Wout x Wout x D.
In all cases, pooling provides some translation invariance which means that an object would be recognizable regardless of where it appears on the frame.
Fully Connected Layer
Neurons in this layer have full connectivity with all neurons in the preceding and succeeding layer as seen in regular FCNN. This is why it can be computed as usual by a matrix multiplication followed by a bias effect.
The FC layer helps to map the representation between the input and the output.

Disadvantages of Pooling layers
Information Loss

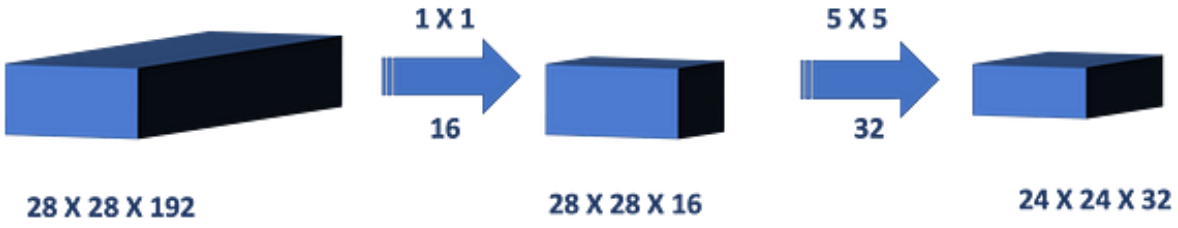
1X1 conv
In 1X1 Convolution simply means the filter is of size 1X1 (Yes — that means a single number as opposed to matrix like, say 3X3 filter). This 1X1 filter will convolve over the ENTIRE input image pixel by pixel.
Staying with our example input of 64X64X3, if we choose a 1X1 filter (which would be 1X1X3), then the output will have the same Height and Weight as input but only one channel — 64X64X1
Now consider inputs with large number of channels — 192 for example. If we want to reduce the depth and but keep the Height X Width of the feature maps (Receptive field) the same, then we can choose 1X1 filters (remember Number of filters = Output Channels) to achieve this effect. This effect of cross channel down-sampling is called ‘Dimensionality reduction’.



Now why would we want to something like that? For that we delve into usage of 1X1 Convolution



Number of Operations : (28X28X32) X (5X5X192) = 120.422 Million Ops



Number of Operations for 1 X 1 Conv Step : (28X28X16) X (1X1X192) = 2.4 Million Ops
Number of Operations for 5 X 5 Conv Step : (28X28X32) X (5X5X16) = 10 Million Ops
Total Number of Operations = 12.4 Million Ops

1X1 Convolution is effectively used for
1. Dimensionality Reduction/Augmentation
2. Reduce computational load by reducing parameter map
3. Add additional non-linearity to the network

4. Create deeper network through “Bottle-Neck” layer
5. Create smaller CNN network which retains higher degree of accuracy

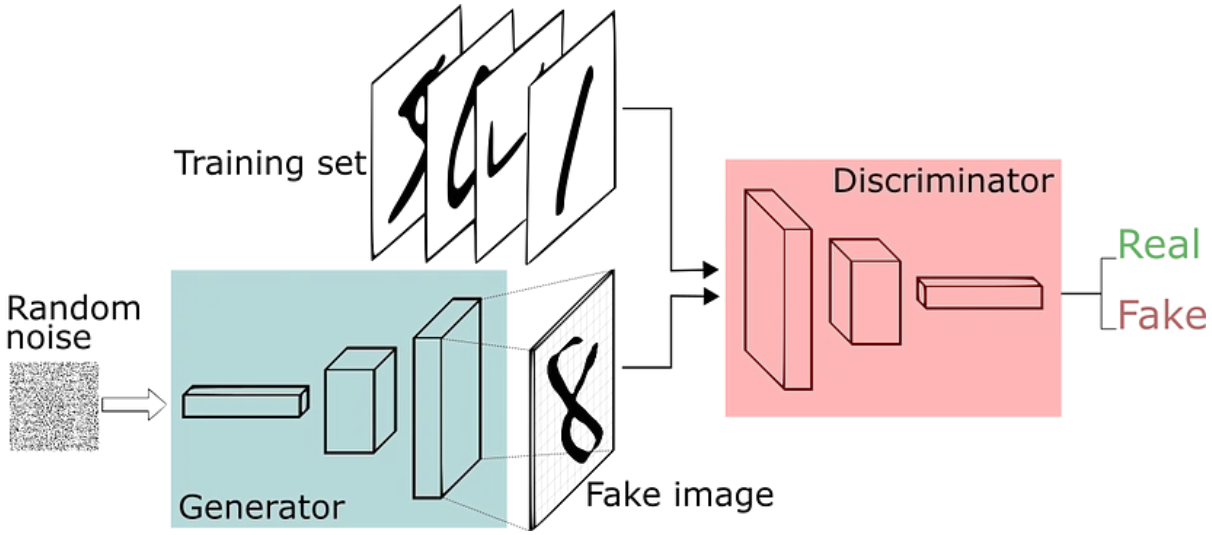
Fully convolutional nets
All layers are either convolution/pooling no linear layers are used.

No. of parameters in CNN example calculation:

SL.No		Activation Shape	Activation Size	# Parameters
1.	Input Layer:	(32, 32, 3)	3072	0
2.	CONV1 (f=5, s=1)	(28, 28, 8)	6272	608
3.	POOL1	(14, 14, 8)	1568	0
4.	CONV2 (f=5, s=1)	(10, 10, 16)	1600	3216
5.	POOL2	(5, 5, 16)	400	0
6.	FC3	(120, 1)	120	48120
7.	FC4	(84, 1)	84	10164
8.	Softmax	(10, 1)	10	850

GAN

Generative Adversarial Networks takes up a game-theoretic approach, unlike a conventional neural network. The network learns to generate from a training distribution through a 2-player game. The two entities are Generator and Discriminator. These two adversaries are in constant battle throughout the training process. Since an adversarial learning method is adopted, we need not care about approximating intractable density functions.



As you can identify from their names, a generator is used to generate real-looking images and the discriminator’s job is to identify which one is a fake. The entities/adversaries are in constant battle as one(generator) tries to fool the other(discriminator), while the other tries not to be fooled. To generate the best images you will need a very good generator and a discriminator. This is because if your generator is not good enough, it will never be able to

fool the discriminator and the model will never converge. If the discriminator is bad, then images which make no sense will also be classified as real and hence your model never trains and in turn you never produces the desired output. The input, random noise can be a Gaussian distribution and values can be sampled from this distribution and fed into the generator network and an image is generated. This generated image is compared with a real image by the discriminator and it tries to identify if the given image is fake or real.

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}}) \right]$$

Since a game-theoretic approach is taken, our objective function is represented as a minimax function. The discriminator tries to maximize the objective function, therefore we can perform gradient ascent on the objective function. The generator tries to minimize the objective function, therefore we can perform gradient descent on the objective function. By alternating between gradient ascent and descent, the network can be trained.

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Gradient Ascent on Discriminator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Gradient Descent on Generator

But when applied, it is observed that optimizing the generator objective function does not work so well, this is because when the sample is generated is likely to be classified as fake, the model would like to learn from the gradients but the gradients turn out to be relatively flat. This makes it difficult for the model to learn. Therefore, the generator objective function is changed as below.

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

New Generator Objective function

Instead of minimizing the likelihood of discriminator being correct, we maximize the likelihood of discriminator being wrong. Therefore, we perform gradient ascent on generator according to this objective function.

Alternate Training approach is followed. Generator weights are untouched while training discriminator and vice-versa

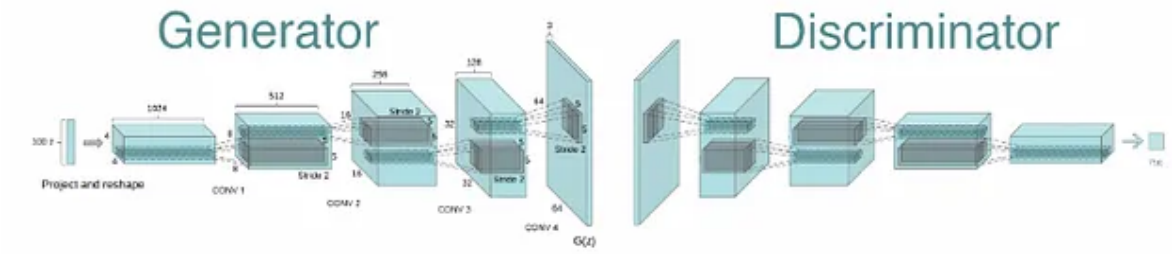
How does a GAN work?

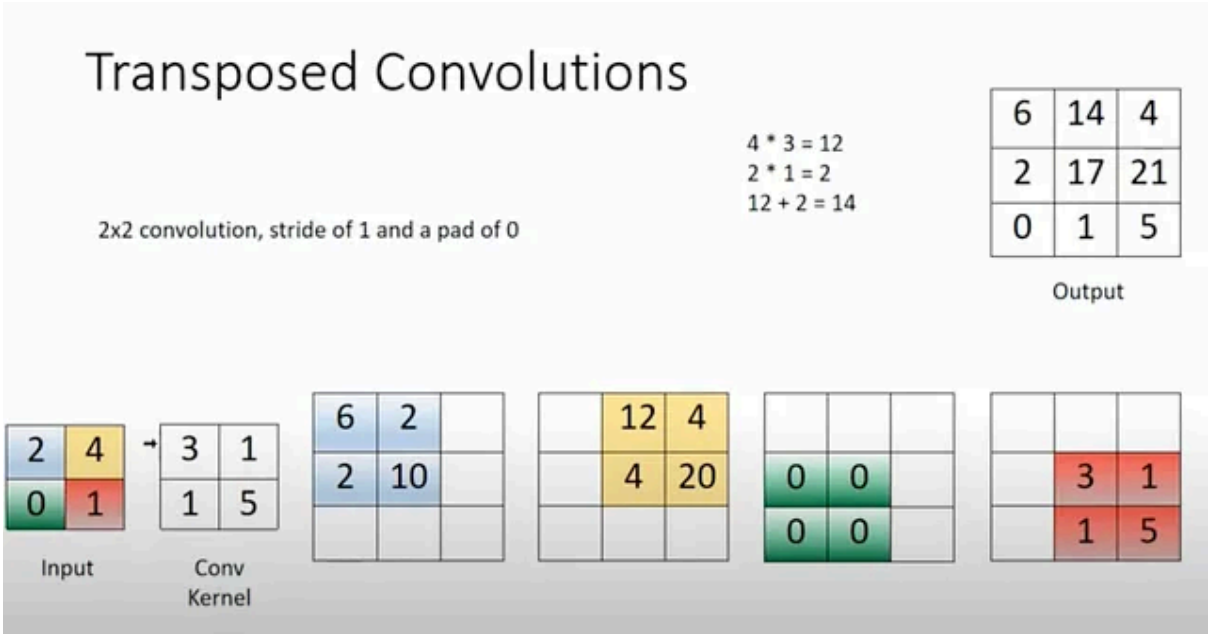
The steps involved in how a GAN works:

1. Initialization: Two neural networks are created: a Generator (G) and a Discriminator (D).
    - G is tasked with creating new data, like images or text, that closely resembles real data.
    - D acts as a critic, trying to distinguish between real data (from a training dataset) and the data generated by G.
2. Generator's First Move: G takes a random noise vector as input. This noise vector contains random values and acts as the starting point for G's creation process. Using its internal layers and learned patterns, G transforms the noise vector into a new data sample, like a generated image.
3. Discriminator's Turn: D receives two kinds of inputs:
    - Real data samples from the training dataset.
    - The data samples generated by G in the previous step. D's job is to analyze each input and determine whether it's real data or something G cooked up. It outputs a probability score between 0 and 1. A score of 1 indicates the data is likely real, and 0 suggests it's fake.
4. The Learning Process: Now, the adversarial part comes in:
    - If D correctly identifies real data as real (score close to 1) and generated data as fake (score close to 0), both G and D are rewarded to a small degree. This is because they're both doing their jobs well.
    - However, the key is to continuously improve. If D consistently identifies everything correctly, it won't learn much. So, the goal is for G to eventually trick D.
5. Generator's Improvement:
    - When D mistakenly labels G's creation as real (score close to 1), it's a sign that G is on the right track. In this case, G receives a significant positive update, while D receives a penalty for being fooled.
    - This feedback helps G improve its generation process to create more realistic data.
6. Discriminator's Adaptation:
    - Conversely, if D correctly identifies G's fake data (score close to 0), but G receives no reward, D is further strengthened in its discrimination abilities.
    - This ongoing duel between G and D refines both networks over time.

As training progresses, G gets better at generating realistic data, making it harder for D to tell the difference. Ideally, G becomes so adept that D can't reliably distinguish real from fake data. At this point, G is considered well-trained and can be used to generate new, realistic data samples.

DCGAN





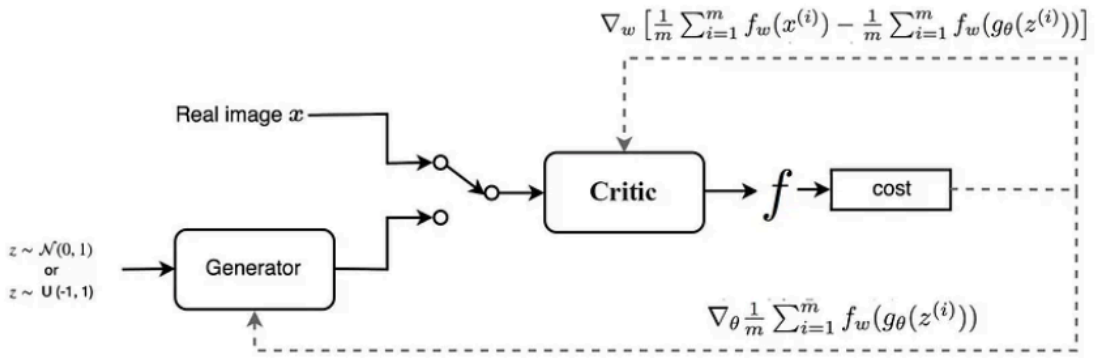
Compared to the original GAN:

1. Replace pooling function with stridden convolutions
2. Get rid of fully connected layers on top of convolutional layers
3. Use LeakyReLu activation

WGAN

During the training, the generator may collapse to a setting where it always produces same outputs. This is a common failure case for GANs, commonly referred to as Mode Collapse

Wasserstein GAN (WGAN) proposes a new cost function using Wasserstein distance that has a smoother gradient everywhere. WGAN learns no matter the generator is performing or not.



The network design is almost the same except the critic does not have an output sigmoid function. The major difference is only on the cost function:

	Discriminator/Critic	Generator
GAN	$\nabla_{\theta_d} \frac{1}{m}\sum_{i=1}^{m}\left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right]$	$\nabla_{\theta_g} \frac{1}{m}\sum_{i=1}^{m} \log\left(D\left(G\left(z^{(i)}\right)\right)\right)$
WGAN	$\nabla_w \frac{1}{m}\sum_{i=1}^{m}\left[f\left(x^{(i)}\right) - f\left(G\left(z^{(i)}\right)\right)\right]$	$\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f\left(G\left(z^{(i)}\right)\right)$

However, there is one major thing missing. f has to be a 1-Lipschitz function. To enforce the constraint, WGAN applies a very simple clipping to restrict the maximum weight value in f, i.e. the weights of the discriminator must be within a certain range controlled by the hyperparameters c.

Conditional GAN

Imagine the need to generate images that are of only Mercedes cars when you have trained your model on a collection of cars. To do that, you need to provide the GAN model with a specific “condition,” which can be done by providing the car’s name (or label). Conditional generative adversarial networks work in the same way as GANs. The generation of data in a CGAN is conditional on specific input information, which could be labels, class information, or any other relevant features. This conditioning enables more precise and targeted data generation.
Architecture and Working of CGANs
Conditioning in GANs:

- GANs can be extended to a conditional model by providing additional information (denoted as y) to both the generator and discriminator.
- This additional information (y) can be any kind of auxiliary information, such as class labels or data from other modalities.

- In the generator, the prior input noise (z) and y are combined in a joint hidden representation.

Generator Architecture:

- The generator takes both the prior input noise (z) and the additional information (y) as inputs.
- These inputs are combined in a joint hidden representation, and the generator produces synthetic samples.
- The adversarial training framework allows flexibility in how this hidden representation is composed.

Discriminator Architecture:

- The discriminator takes both real data (x) and the additional information (y) as inputs.
- The discriminator's task is to distinguish between real data and synthetic data generated by the generator conditioned on y.

Loss Function:

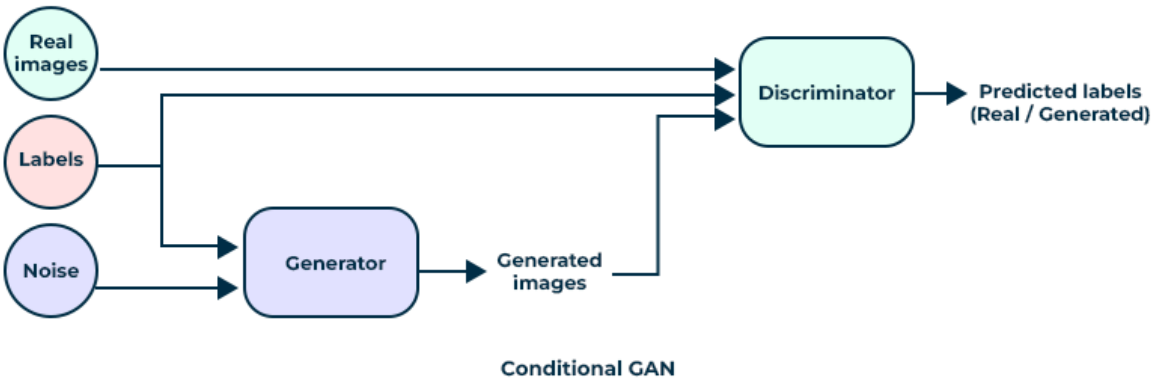
The objective function for the conditional GAN is formulated as a two-player minimax game:

$$min_G max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[logD(x|y)] + \mathbb{E}_{z \sim p_z}(z)[log(1 - D(G(zy)))]$$

Here,

- $\mathbb{E}$ represents the expected operator. It is used to denote the expected value of a random variable. In this context, $\mathbb{E}_{x \sim p_{data}}$ represents the expected value with respect to the real data distribution $p_{data}(x)$, and $\mathbb{E}_{z \sim p_z}(z)$ represents the expected value with respect to the prior noise distribution $p_z(z)$.
- The objective is to simultaneously minimize the generator's ability to fool the discriminator and maximize the discriminator's ability to correctly classify real and generated samples.
- The first term $(logD(xy))$ encourages the discriminator to correctly classify real samples.
- The second term $(log(1D(G(zy))))$ encourages the generator to produce samples that are classified as real by the discriminator.

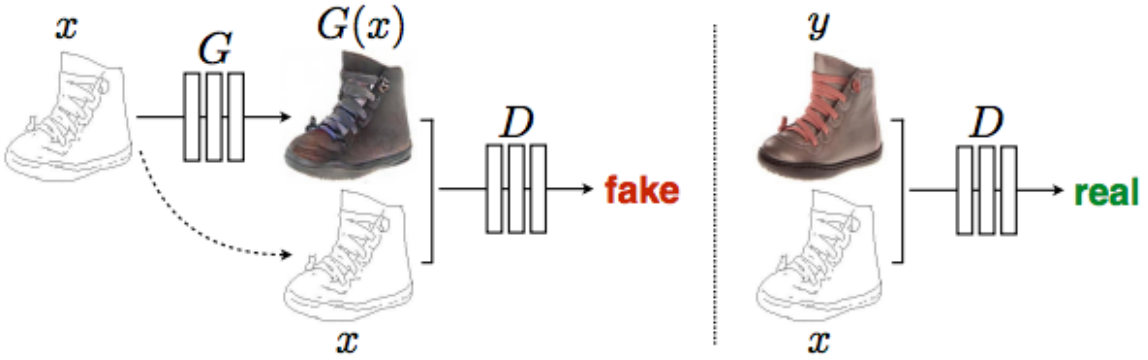
This formulation creates a balance in which the generator improves its ability to generate realistic samples, and the discriminator becomes more adept at distinguishing between real and generated samples conditioned on y.



Pix2Pix

Image-to-Image translation is another example of a task which Generative Adversarial Networks (GANs) are perfectly suited for. These are tasks in which it is nearly impossible to hard-code a loss function for. Most studies on GANs are concerned with novel image synthesis, translating from a random vector z into an image. Image-to-Image translation converts one image to another such as the edges of the bag above to the photo image. Another interesting example of this is shown below:

Interesting idea of translating from satellite imagery into a Google Maps-style street view Image-to-Image translation is also useful in applications such as colorization and super-resolution. However, many of the implementation ideas specific to the pix2pix algorithm are also relevant for those studying novel image synthesis.



A very high-level view of the Image-to-Image translation architecture in this paper is depicted above. Similar to many image synthesis models, this uses a Conditional-GAN framework. The conditioning image, x is applied as the input to the generator and as input to the discriminator.

Dual Objective Function with Adversarial and L1 Loss

A naive way to do Image-to-Image translation would be to discard the adversarial framework altogether. A source image would just be passed through a parametric function and the difference in the resulting image and the ground truth output would be used to update the weights of the network. However, designing this loss function with standard distance measures such as L1 and L2 will fail to capture many of the important distinctive characteristics between these images. However, the authors do find some value to the L1 loss function as a weighted sidekick to the adversarial loss function.

The Conditional-Adversarial Loss (Generator versus Discriminator) is very popularly formatted as follows:

$$\mathcal{L}_{cGAN}(G,D) = \mathbb{E}_{x,y}[\log D(x,y)] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x,z))],$$

The L1 loss function previously mentioned is shown below:

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x,z)\|_1].$$

Combining these functions results in:

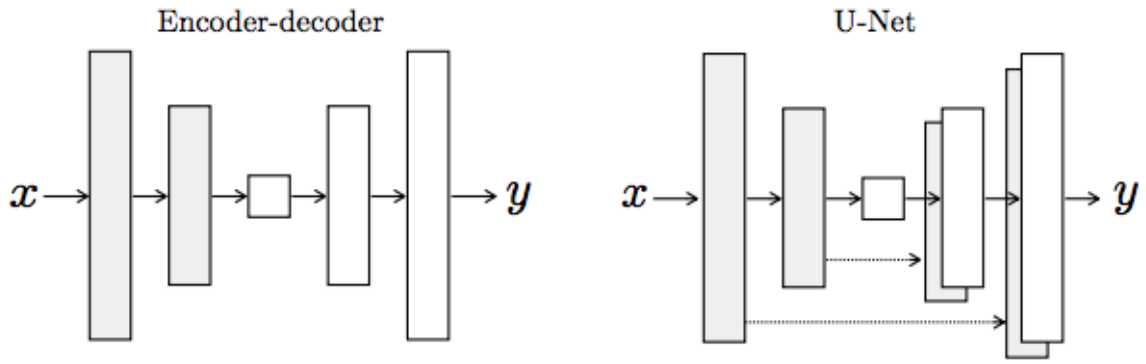
$$G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G,D) + \lambda \mathcal{L}_{L1}(G).$$

In the experiments, the authors report that they found the most success with the lambda parameter equal to 100.

U-Net Generator

The U-Net architecture used in the Generator of the GAN was a very interesting component of this paper. Image Synthesis architectures typically take in a random vector of size 100x1, project it into a much higher dimensional vector with a fully connected layer, reshape it, and

then apply a series of de-convolutional operations until the desired spatial resolution is achieved. In contrast, the Generator in pix2pix resembles an auto-encoder.



The Skip Connections in the U-Net differentiate it from a standard Encoder-decoder architecture

The Generator takes in the Image to be translated and compresses it into a low-dimensional, “Bottleneck”, vector representation. The Generator then learns how to upsample this into the output image. As illustrated in the image above, it is interesting to consider the differences between the standard Encoder-Decoder structure and the U-Net. The U-Net is similar to ResNets in the way that information from earlier layers are integrated into later layers. The U-Net skip connections are also interesting because they do not require any resizing, projections etc. since the spatial resolution of the layers being connected already match each other.

PatchGAN Discriminator

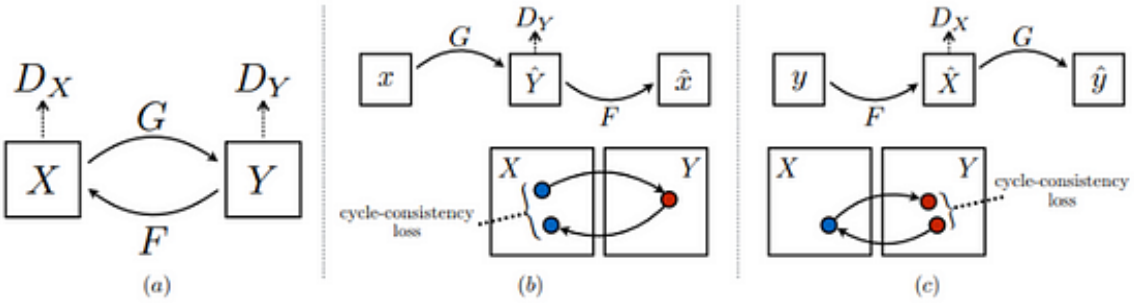
The PatchGAN discriminator used in pix2pix is another unique component to this design. The PatchGAN / Markovian discriminator works by classifying individual (N x N) patches in the image as “real vs. fake”, opposed to classifying the entire image as “real vs. fake”. The authors reason that this enforces more constraints that encourage sharp high-frequency detail. Additionally, the PatchGAN has fewer parameters and runs faster than classifying the entire image. The image below depicts results experimenting with the size of N for the N x N patches to be classified:

The 70 x 70 Patch is found to produce the best results

CycleGAN

CycleGAN is a powerful deep-learning architecture that enables the task of image-to-image translation without the need for paired training data. It uses two generative adversarial networks (GANs) to learn the mapping between two domains of images. The model is trained to capture the characteristics of the target domain and to generate new images from the source domain that share the same characteristics.

The model is initialized by training the two GANs separately. First, the generator of the first GAN is trained to generate a target domain image from a source domain image. The discriminator of the first GAN is trained to identify real target domain images from generated target domain images. Meanwhile, the generator of the second GAN is trained to generate a source domain image from a target domain image.
The discriminator of the second GAN is trained to detect real source domain images from generated source domain images. Once the two GANs have been trained separately, the CycleGAN is defined.
The generators of the two GANs form the generators of the CycleGAN, and the discriminators of the two GANs form the discriminators of the CycleGAN. The model is then trained to generate an image from the source domain that is similar to an image from the target domain and vice versa.
The CycleGAN is trained using a cycle-consistent loss, which encourages the model to generate images that are indistinguishable from real images from the target domain. The model is further improved by adding an identity mapping component. This component helps to preserve the characteristics of the source domain images when they are translated to the target domain.



CycleGAN has several key advantages over other image-to-image translation models

1. Accurate — CycleGAN is more accurate than its predecessors because it takes advantage of unpaired data, meaning that it can produce better results without the need for a large number of paired training images. This is important because paired training images are often difficult to obtain.
2. Robust — CycleGAN is more robust to domain shifts in the data, meaning that it can perform well even if the input images come from different domains. This is important because it allows CycleGAN to be used for a range of image-to-image translation tasks, such as changing the style of an image, translating from one domain to another, or even translating from one language to another.
3. Small dataset — CycleGAN can be used to generate high-quality images with relatively little training data. This is important for tasks in which training data is limited, such as medical image-to-image translation.
4. Easy to train — CycleGAN is easy to train and use, meaning that it is accessible to practitioners who may not have access to the complex infrastructure required for other GAN models.

Overall, CycleGAN is an incredibly powerful and versatile tool for image-to-image translation, and its advantages over other models make it an ideal choice for practitioners looking to produce high-quality results without requiring large amounts of training data or complex infrastructure

There are several disadvantages to using a CycleGAN:

1. It is difficult to control the output of the cycle gang, which can lead to results that are less than desirable.
2. The CycleGAN can be slow to train, particularly when there are a large number of training images.
3. The CycleGAN can be prone to overfitting, which can lead to poor results on unseen data.
4. The CycleGAN can be difficult to interpret, making it hard to understand why the model is generating the results it is.
5.

- **Adversarial Loss:** We apply adversarial loss to both our mappings of generators and discriminators. This adversary loss is written as :

$$Loss_{advers}(G, D_y, X, Y) = \frac{1}{m}\sum (1 - D_y(G(x)))^2$$
$$Loss_{advers}(F, D_x, Y, X) = \frac{1}{m}\sum (1 - D_x(F(y)))^2$$

- **Cycle Consistency Loss**: Given a random set of images adversarial network can map the set of input image to random permutation of images in the output domain which may induce the output distribution similar to target distribution. Thus adversarial mapping cannot guarantee the input x_i to y_i . For this to happen the author proposed that process should be cycle-consistent. This loss function used in Cycle GAN to measure the error rate of inverse mapping G(x) -> F(G(x)). The behavior induced by this loss function cause closely matching the real input (x) and F(G(x))

$$Loss_{cyc}(G, F, X, Y) = \frac{1}{m}\left[(F(G(x_i)) - x_i) + (G(F(y_i)) - y_i)\right]$$

The Cost function we used is the sum of adversarial loss and cyclic consistent loss:

$$L(G, F, D_x, D_y) = L_{advers}(G, D_y, X, Y) + L_{advers}(F, D_x, Y, X) + \lambda L_{cycl}(G, F, X, Y)$$

and our aim is :

$$\underset{G,F}{argmin}\ \underset{D_x,D_y}{max}\ L(G, F, D_x, D_y)$$

- **Photo Generation from Painting**: CycleGAN can also be used to transform photo from paintings and vice-versa. However to improve this transformation., the authors also introduced an additional loss called Identity loss. This loss can be defined as :

$$L_{identity}(G, F) = \mathbb{E}_{y\ p(y)}\left[\|G(y) - y\|_1\right] + \mathbb{E}_{x\ p(x)}\left[\|F(x) - x\|_1\right]$$

SRGAN

This paper presents SRGAN, a generative adversarial network (GAN) for image super resolution (SR).
Super resolution is a task of estimating a high resolution (HR) image from its low resolution (LR) counterpart. This estimated image is called as super resolved (SR) image.
Previous super resolution methods are mainly driven by the choice of the optimization function. Most commonly used optimization target for the supervised SR algorithms is the minimization of the mean-squared error (MSE) and maximization of peak signal-to-noise(PSNR) which are defined based on the pixel-wise image differences. The

PSNR ratios obtained from these methods is high but the images are perceptually not satisfying.

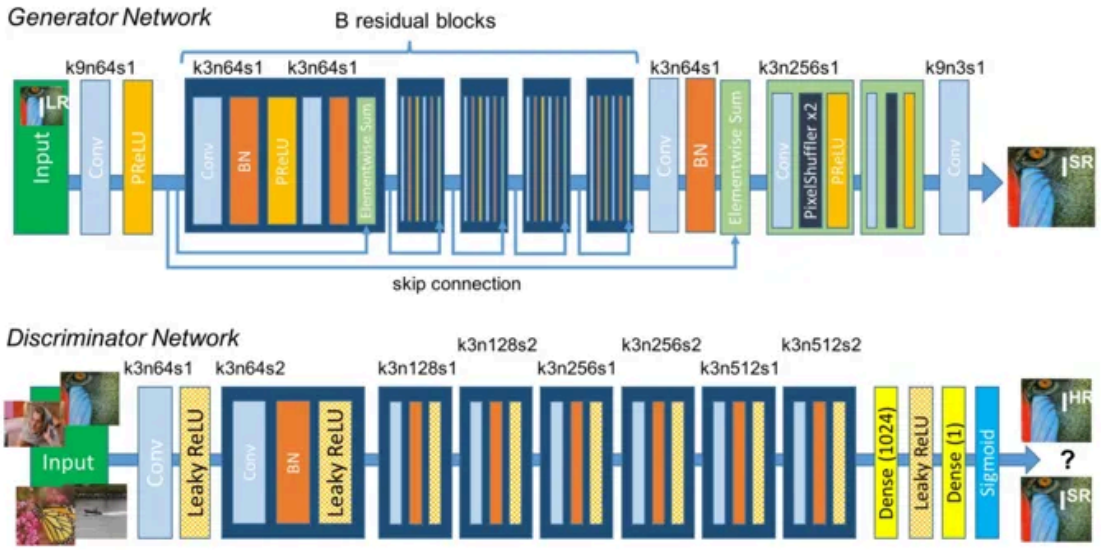


Figure 4: Architecture of Generator and Discriminator Network with corresponding kernel size (k), number of feature maps (n) and stride (s) indicated for each convolutional layer.

The model consists of 2 main blocks:

1. Generator
2. Discriminator

In training, a low resolution image (ILR) is obtained by applying a gaussian filter to a high resolution image (IHR) followed by a down-sampling operation with down-sampling factor r.

•ILR is given as : W x H x C

•IHR and ISR are given as : rW x rH x C

1.Generator function G estimates for a given input LR image its corresponding HR image which is a super resolved image SR.

2.Discriminator D is trained to distinguish super resolved images and real images.

Loss Function

Perceptual loss lsr is defined as the weighted sum of a content loss and an adversarial component:

$$l^{SR} = \underbrace{\underbrace{l_{\mathrm{X}}^{SR}}_{\text{content loss}} + \underbrace{10^{-3} l_{Gen}^{SR}}_{\text{adversarial loss}}}_{\text{perceptual loss (for VGG based content losses)}}$$

Content Loss

1.MSE loss: Pixel-wise error between high resolution image and super resolved image (generated image) is given as below:

$$l_{MSE}^{SR} = \frac{1}{r^2 W H} \sum_{x=1}^{rW} \sum_{y=1}^{rH} (I_{x,y}^{HR} - G_{\theta_G}(I^{LR})_{x,y})^2$$

2.VGG loss: φ(i,j) indicates the feature map obtained by the j-th convolution (after activation) before the i-th maxpooling layer within the VGG19 network. VGG loss is defined as the euclidean distance between the feature representations of a reconstructed image and the reference image:

$$l^{SR}_{VGG/i.j} = \frac{1}{W_{i,j}H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(I^{HR})_{x,y} - \phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y})^2$$

Adversarial loss

The generative loss is defined based on the probabilities of the discriminator over all training samples (N) :

$$l^{SR}_{Gen} = \sum_{n=1}^{N} -\log D_{\theta_D}(G_{\theta_G}(I^{LR}))$$

The discriminator is trained to solve the maximization problem in the following equation:

$$\min_{\theta_G} \max_{\theta_D} \mathbb{E}_{I^{HR} \sim p_{\text{train}}(I^{HR})}[\log D_{\theta_D}(I^{HR})] + \mathbb{E}_{I^{LR} \sim p_G(I^{LR})}[\log(1 - D_{\theta_D}(G_{\theta_G}(I^{LR}))]$$

The Generator is trained to minimize the following equation:

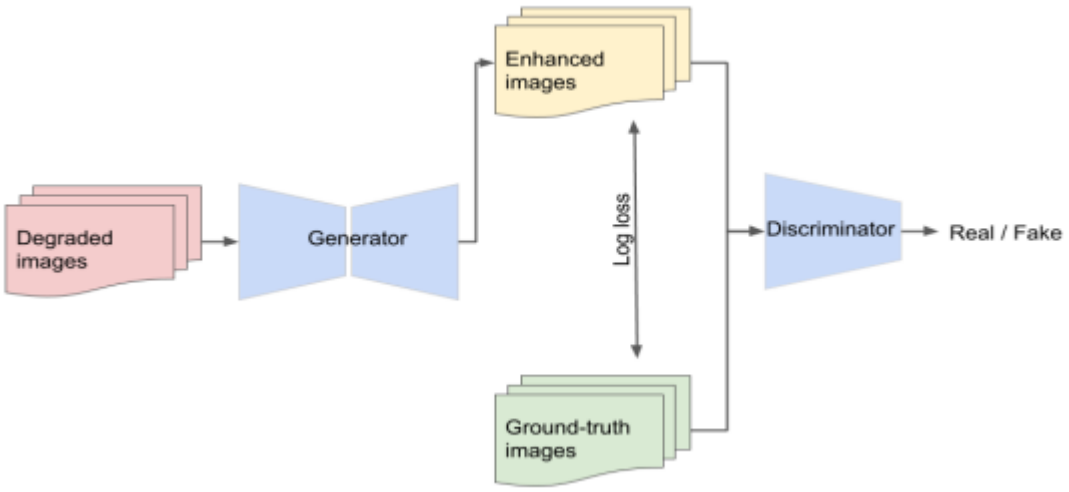
$$\hat{\theta}_G = \arg\min_{\theta_G} \frac{1}{N} \sum_{n=1}^{N} l^{SR}(G_{\theta_G}(I^{LR}_n), I^{HR}_n)$$

Where lsr, Perceptual loss is the sum of content loss and adversarial loss described earlier:

$$l^{SR} = \underbrace{\underbrace{l^{SR}_{\text{X}}}_{\text{content loss}} + \underbrace{10^{-3} l^{SR}_{Gen}}_{\text{adversarial loss}}}_{\text{perceptual loss (for VGG based content losses)}}$$

Here, $D_{\theta_D}(G_{\theta_G}(I^{LR}))$ is the probability that the reconstructed image $G_{\theta_G}(I^{LR})$ is a natural HR image. For better gradient behavior we minimize $-\log D_{\theta_D}(G_{\theta_G}(I^{LR}))$ instead of $\log[1 - D_{\theta_D}(G_{\theta_G}(I^{LR}))]$ [22].

DEGAN



$$L_{GAN}(\varphi_G, \varphi_D) = \mathbb{E}_{I^W, I^{GT}} \log[D_{\varphi_D}(I^W, I^{GT})] + \mathbb{E}_{I^W} \log[1 - D_{\varphi_D}(I^W, G_{\varphi_G}(I^W))]$$

$$L_{log}(\varphi_G) = \mathbb{E}_{I^{GT}, I^W}[-(I^{GT} \log(G_{\varphi_G}(I^W)) + ((1 - I^{GT}) \log(1 - G_{\varphi_G}(I^W)))]$$

$$L_{net}(\varphi_G, \varphi_D) = min_{\varphi_G} max_{\varphi_D} L_{GAN}(\varphi_G, \varphi_D) + \lambda L_{log}(\varphi_G)$$

GCN

Optimizers

Activation Functions