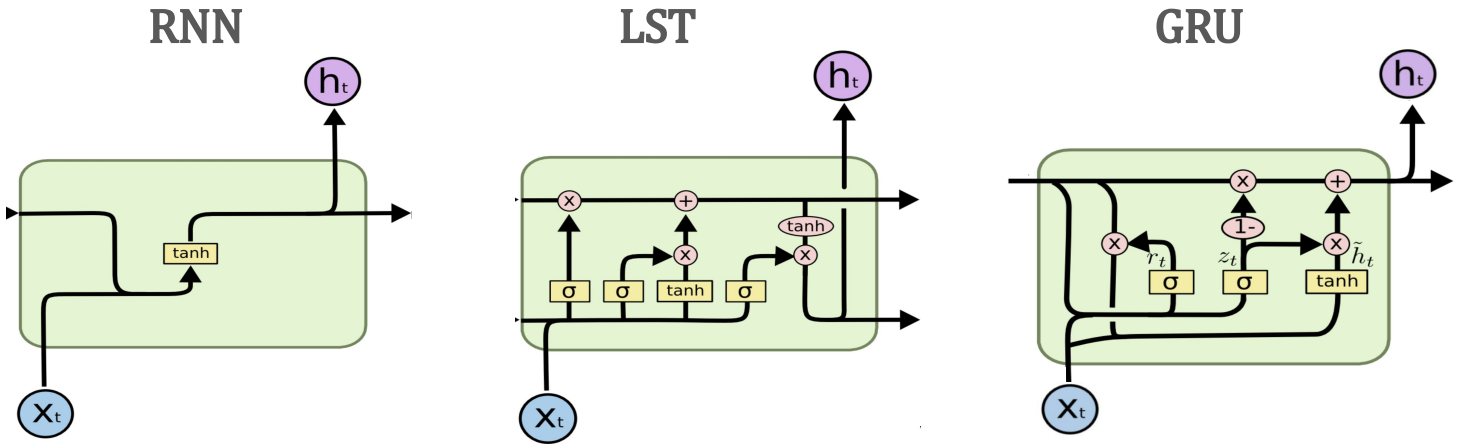
APS360: Applied Fundamentals of Deep Learning

Week 10: Transformers

RNNs

Vanilla RNNs, LSTMs, and GRUs are **sequential in nature**

They are **inefficient** as we cannot take full advantage of vectorization and parallel GPUs



Attention

When humans read or look:

- They focus (**attend**) on some regions within the input → **high resolution**
- They pay less attention to surrounding and perceive it less → **low resolution**

Circles → fixation at each location

Lines → participant's eye movements from one location to another

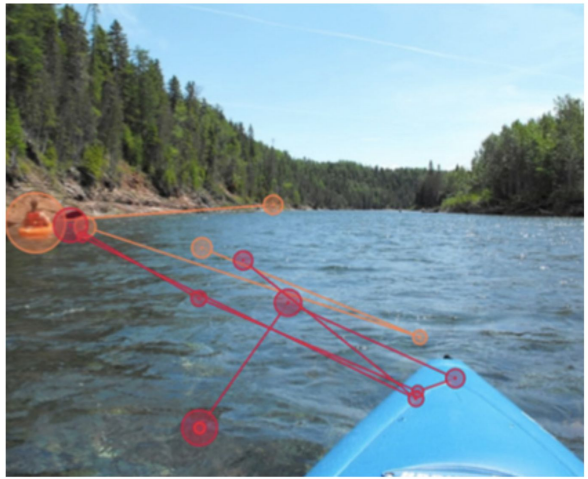


Image courtesy Visual attention

Image courtesy UX Matters

Users re-read the more difficult questions

Simple Attention Mechanism

Let's design an attention-based pooling for **classifying tweets**

We want to use it to pool the word embeddings of a tweet into a **tweet embedding**

Of course, we can simply **sum** or **average** the word embeddings

But then it assumes that all the words have a same importance

Can we do better?

Simple Attention Mechanism

We can use a fully-connected network that takes in word embeddings and generates a **single score for each embedding**

We can then **normalize** these scores across all words within the tweet using a **softmax**

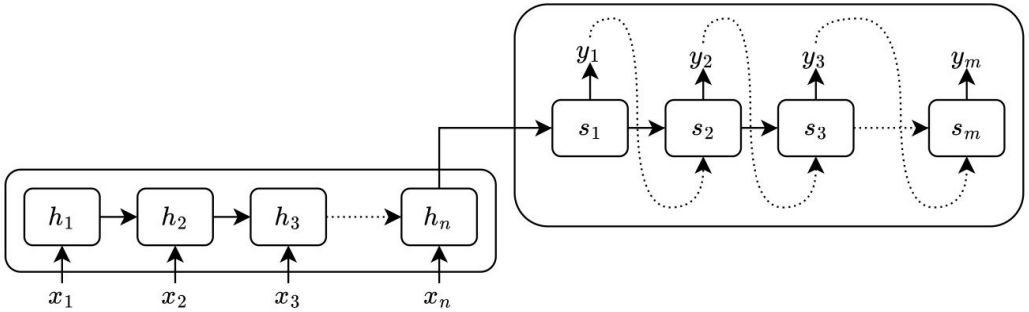
We then multiply each embedding with its normalized score and sum them up:

$$c_i = \sum_j \alpha_{ij} h_j$$

This network is trained end-to-end with the classifier

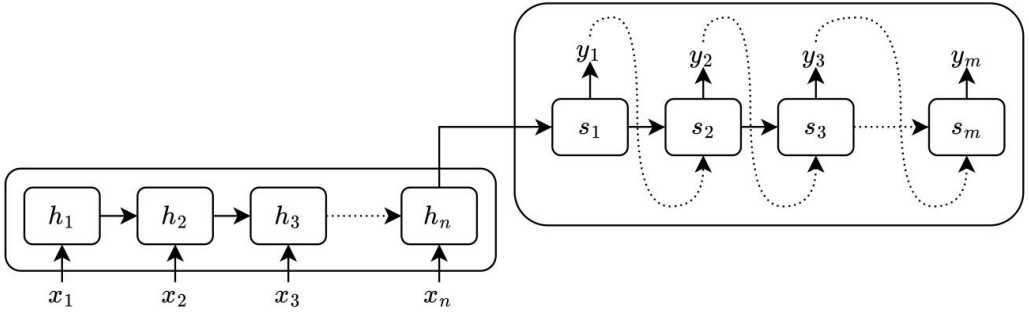
Attention in RNNs

RNN without Attention

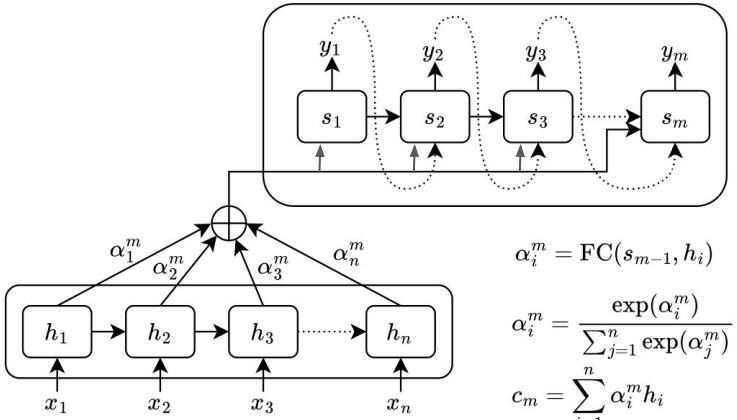


Attention in RNNs

RNN without Attention



RNN with Attention (cross-attention)



$$\alpha_i^m = \mathrm{FC}(s_{m-1}, h_i)$$

$$\alpha_i^m = \frac{\exp(\alpha_i^m)}{\sum_{j=1}^{n} \exp(\alpha_j^m)}$$

$$c_m = \sum_{i=1}^{n} \alpha_i^m h_i$$

$$s_m = \mathrm{Update}(s_{m-1}, c_m, y_{m-1})$$

How to compute attention score?

Suppose we have two embeddings $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$

We can use different methods to compute attention score between them:

- Dot product $\text{score}(\mathbf{a},\mathbf{b}) = \mathbf{a}^T.\mathbf{b}$
- Cosine similarity $\text{score}(\mathbf{a},\mathbf{b}) = \mathbf{a}^T.\mathbf{b}/||\mathbf{a}||.||\mathbf{b}||$
- Bilinear $\text{score}(\mathbf{a},\mathbf{b}) = \mathbf{a}^T\mathbf{W}\mathbf{b}$
- MLP $\text{score}(\mathbf{a},\mathbf{b}) = \text{Sigmoid}(\mathbf{W}[\mathbf{a};\mathbf{b}])$
- ……

Transformers

Attention in Transformers

Transformers are a class of deep models that are based on **self-attention** where attention is modeled as a neural dictionary:

- It retrieves a **value** v_i for a **query** q based on a **key** k_i
- Values, queries, and keys are d-dimensional embeddings
- However, rather than retrieving a single value for a query, it uses a **soft retrieval**
- It retrieves all the values but then computes their importance w.r.t query based on the similarity between the query and their keys

$$\text{attention}(q, \mathbf{k}, \mathbf{v}) = \sum_i \text{similarity}(q, k_i) \times v_i$$

Attention in Transformers

Suppose X is an input sequence consisting of n tokens where each token $t \in \mathbb{R}^i$

To compute the queries, keys, and values from the X, we use three linear layers:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q,\ \mathbf{X} \in \mathbb{R}^{n \times i},\ \mathbf{W}_Q \in \mathbb{R}^{i \times k},\ \mathbf{Q} \in \mathbb{R}^{n \times k}$$
$$\mathbf{K} = \mathbf{X}\mathbf{W}_K,\ \mathbf{X} \in \mathbb{R}^{n \times i},\ \mathbf{W}_K \in \mathbb{R}^{i \times k},\ \mathbf{K} \in \mathbb{R}^{n \times k}$$
$$\mathbf{V} = \mathbf{X}\mathbf{W}_V,\ \mathbf{X} \in \mathbb{R}^{n \times i},\ \mathbf{W}_V \in \mathbb{R}^{i \times v},\ \mathbf{V} \in \mathbb{R}^{n \times v}$$

Note that queries, keys, and values are computed on the same input sequence

Attention in Transformers

Self-attention in Transformers is defined as follows:

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

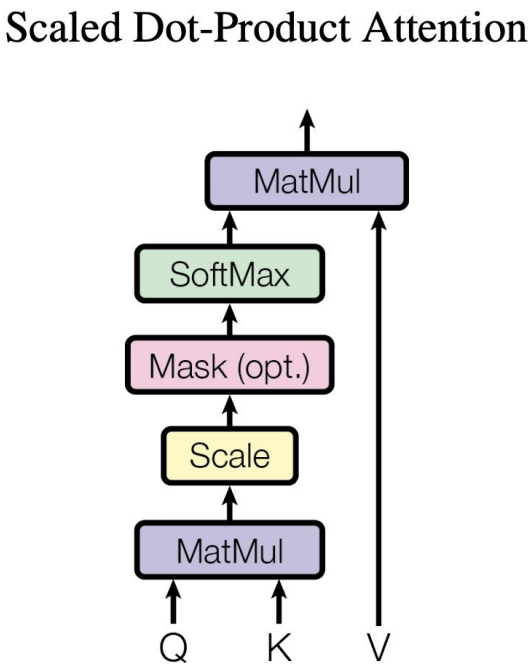


Image courtesy Attention Is All You Need

Attention in Transformers

Self-attention in Transformers is defined as follows:

Pair-wise attention score between all tokens within the input sequence, $\in \mathbb{R}^{n \times n}$

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

Hidden dimension of Q, K is used to keep the scores from blowing up for large d_k

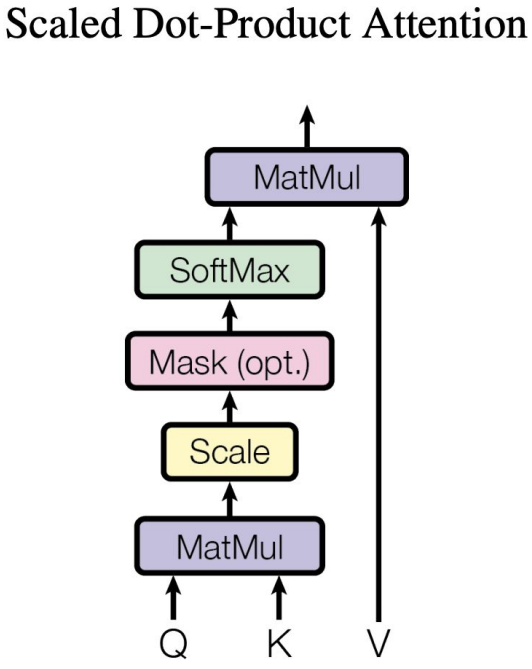


Image courtesy Attention Is All You Need

Attention in Transformers

Self-attention in Transformers is defined as follows:

$$\text{Attention}(Q, K, V) = \overbrace{\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V}^{\in \mathbb{R}^{n \times v}}$$

New representation of each token based on weighted combination of other tokens (contextual representations)

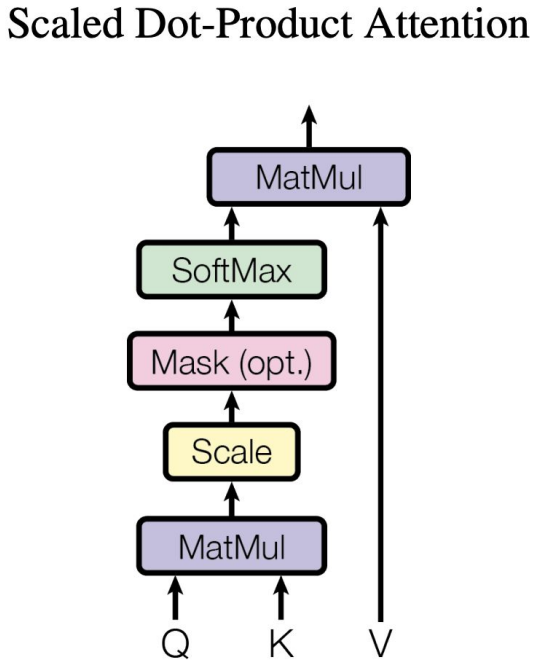


Image courtesy Attention Is All You Need

Multi-Head Attention

To improve the performance:

1. Divide the representation space to h sub-spaces
2. Run parallel linear layers and attentions
3. Concatenate them back to form the original space

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\textbf{where } \text{head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

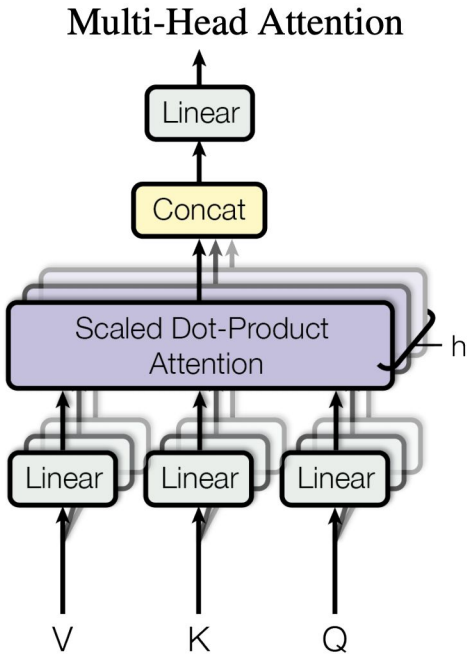


Image courtesy Attention Is All You Need

Transformer Encoders

Each encoder layer consists of:

1. A multi-head self-attention sub-layer
2. A fully-connected sub-layer
3. A residual connection around each of the two sub-layers followed by layer normalization

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

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

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

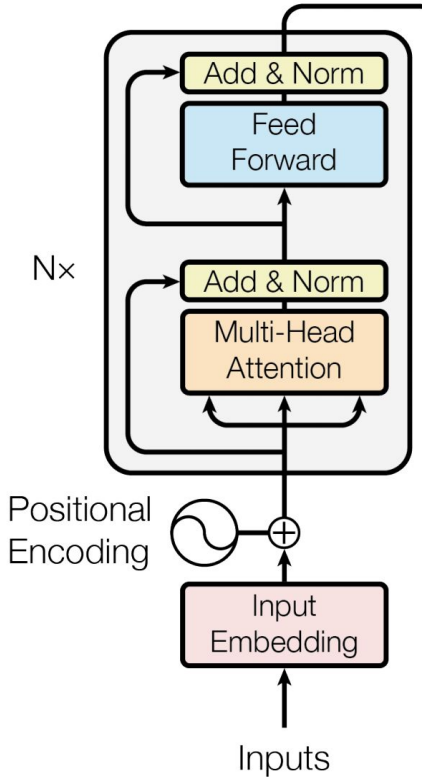


Image courtesy Attention Is All You Need

Positional Encoding

The model does **not have recurrent or convolutional layers** so it doesn't take into account the order of sequence.

We use **positional encoding** to make use of the order which allows the model to easily learn to attend by relative positions.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

position dimension

pos d_{model}=4, /100 instead of 10000

	pos				
I	0	P_{00}=sin(0) = 0	P_{01}=cos(0) = 1	P_{02}=sin(0) = 0	P_{03}=cos(0) = 1
am	1	P_{10}=sin(1/1) = 0.84	P_{11}=cos(1/1) = 0.54	P_{12}=sin(1/10) = 0.10	P_{13}=cos(1/10) = 1.0
a	2	P_{20}=sin(2/1) = 0.91	P_{21}=cos(2/1) = -0.42	P_{22}=sin(2/10) = 0.20	P_{23}=cos(2/10) = 0.98
Robot	3	P_{30}=sin(3/1) = 0.14	P_{31}=cos(3/1) = -0.99	P_{32}=sin(3/10) = 0.30	P_{33}=cos(3/10) = 0.96

RNNs Vs. Transformers

RNN	Transformer
• Struggling with long range dependencies • Gradient vanishing and explosion • Large number of training steps • Recurrence prevents parallel computation	• Facilitate long range dependencies • Less likely to have gradient vanishing and explosion problem • Fewer training steps • No recurrence, facilitates parallel computation

Content from CS480/680 Spring 2019 Pascal Poupart

PyTorch Implementation

PyTorch: Encoder

```
class TransformerEncoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(TransformerEncoder, self).__init__()
        self.linear_q = nn.Linear(input_size, hidden_size)
        self.linear_k = nn.Linear(input_size, hidden_size)
        self.linear_v = nn.Linear(input_size, hidden_size)
        self.linear_x = nn.Linear(input_size, hidden_size)
        self.attention = nn.MultiheadAttention(hidden_size, num_heads=4, batch_first=True)
        self.fc = nn.Sequential(
                        nn.Linear(hidden_size, hidden_size),
                        nn.ReLU(),
                        nn.Linear(hidden_size, hidden_size))
        self.norm = nn.LayerNorm(hidden_size)

    def forward(self, x):
        q, k, v = self.linear_q(x), self.linear_k(x), self.linear_v(x)
        x = self.norm(self.linear_x(x) + self.attention(q, k, v))
        x = self.norm(x + self.fc(x))
        return x
```

PyTorch: Classifier

```
class TweetTransformer(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetTransformer, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.encoder = TransformerEncoder(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x, pos):
        # Add GloVe vectors to positional encoding
        x = self.emb(x) + pos
        x = self.encoder(x)
        # Add embeddings from transformer encoding to get tweet embedding
        x = torch.sum(x, -1)
        # Classify
        return self.fc(x)
```

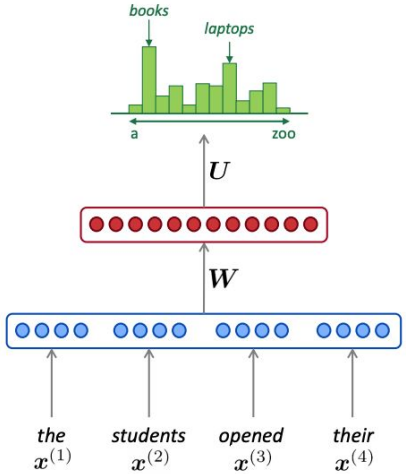
Transformers for Language Modeling

Language Modeling

We can use a self-supervised objective such as predicting the next work to learn embeddings over tokens:

Word2Vec/GloVe :

- Learn static embeddings
- One embedding for all senses



Language Modeling

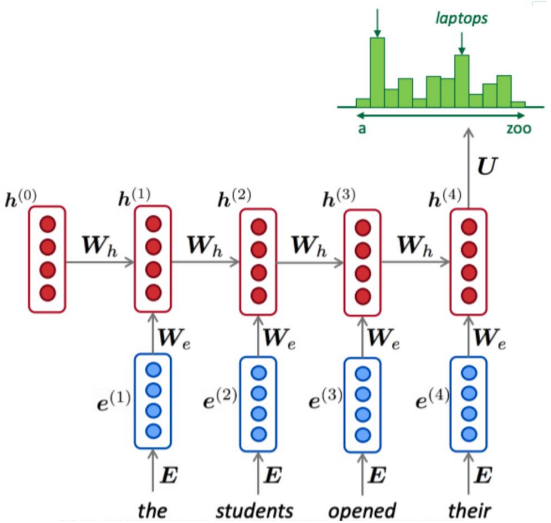
We can use a self-supervised objective such as predicting the next work to learn embeddings over tokens:

Word2Vec/GloVe :

- Learn static embeddings
- One embedding for all senses

RNNs/Transformers :

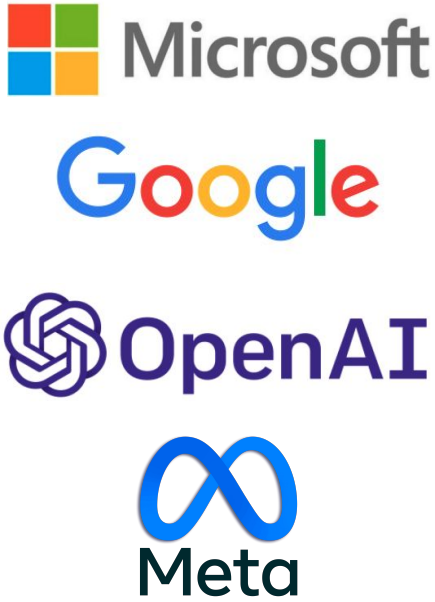
- Learn contextual embeddings
- Embedding of a same word changes according to the sentence it appear in



Large Language Models with Transformers

Transformer models differ in terms of their prediction tasks and training objectives

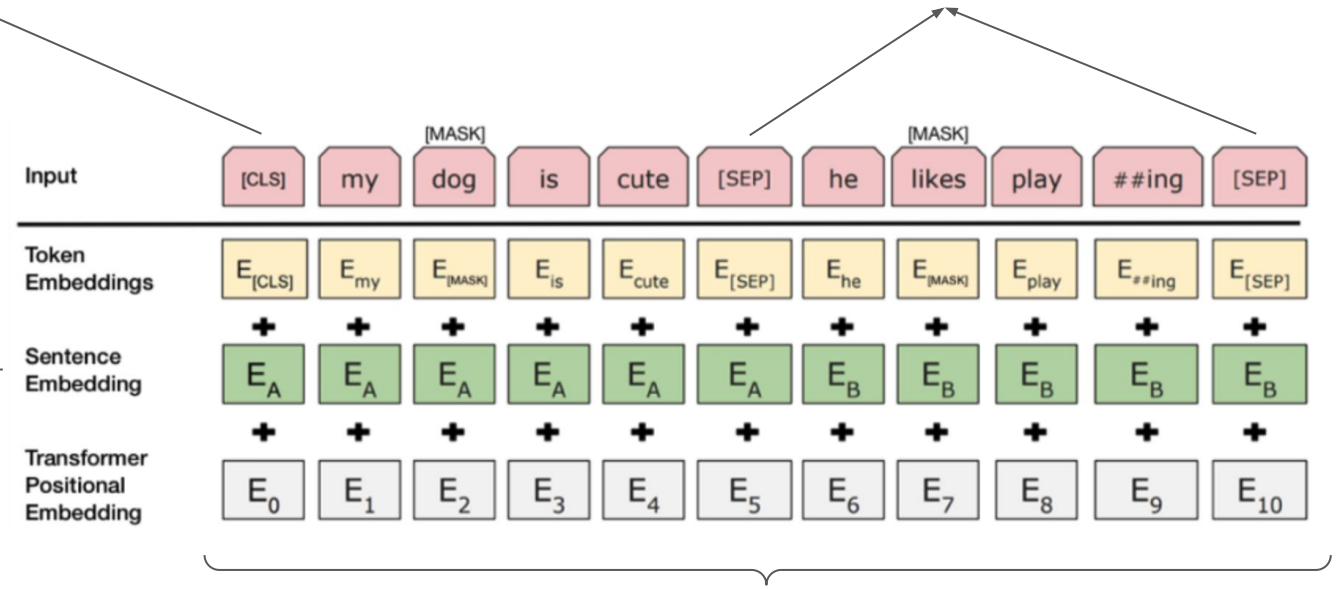
- Llama
- Gemini
- GPT4
- Claude
- BERT



Input Embeddings

`[CLS]` → token always appears at the start of the text, and is specific to classification tasks.

`[SEP]` → marks the end of a sentence, or the separation between two sentences



Specifies each token belongs to which sentence: sentence 0 (vector of 0s) or sentence 1 (vector of 1s)

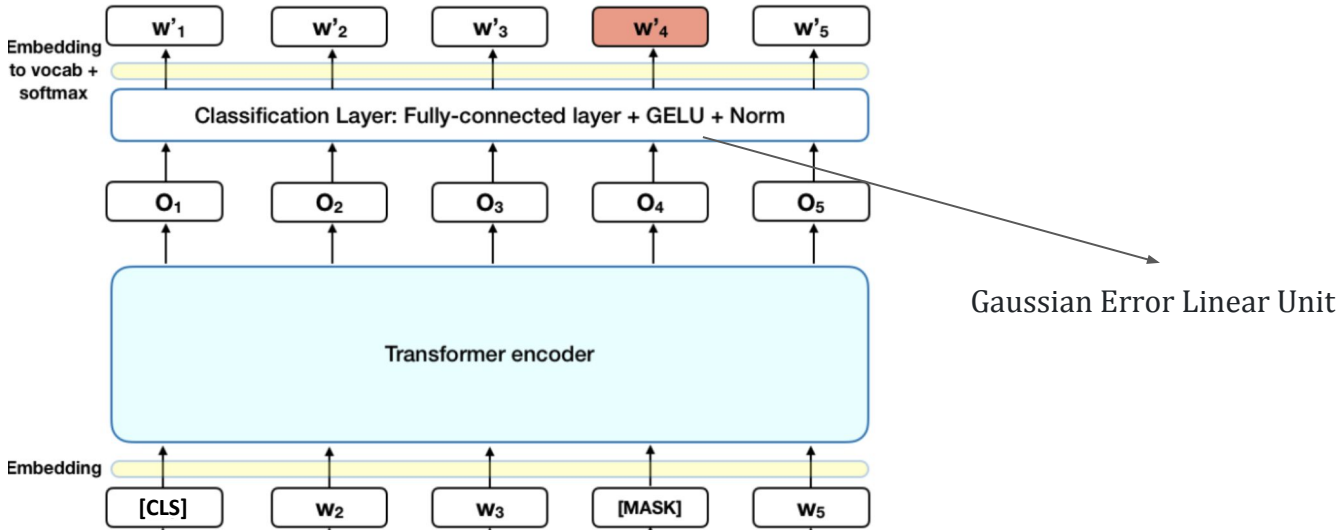
Element-wise sum

Task 1: Masked Word Prediction

Replace **15% of words** , at random, with `[MASK]` token

Using the context of non-masked words, **predict original value** of `[MASK]` token

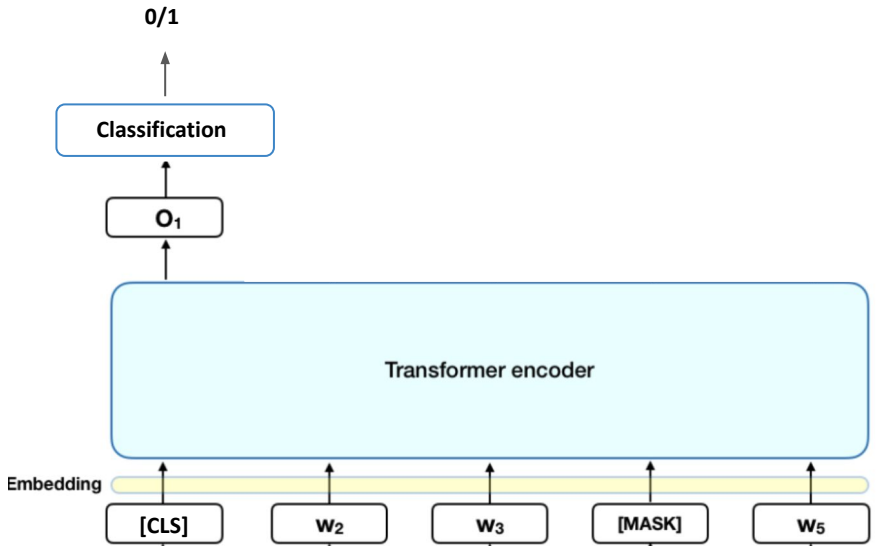
Loss is computed on just the masked word (contrast with next word prediction)



Task 2: Next Sentence Prediction

Given two sentences, predict if they **appear together** .

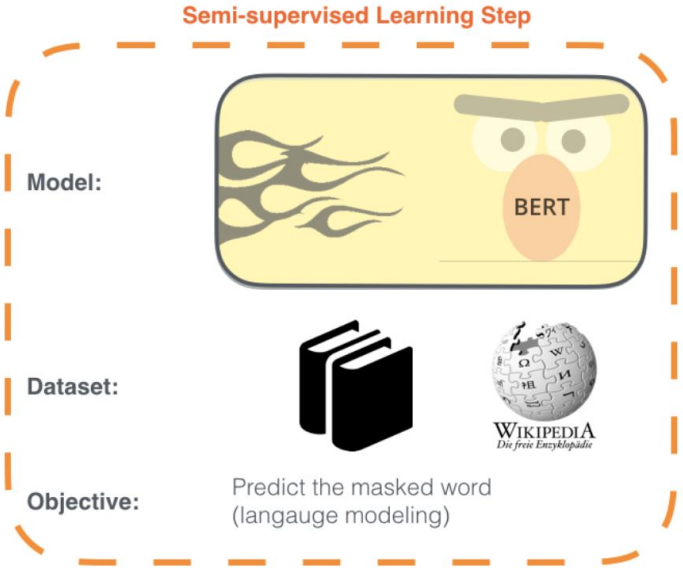
Create 50% positive and 50% negative pairs of sentences (≤512 tokens)



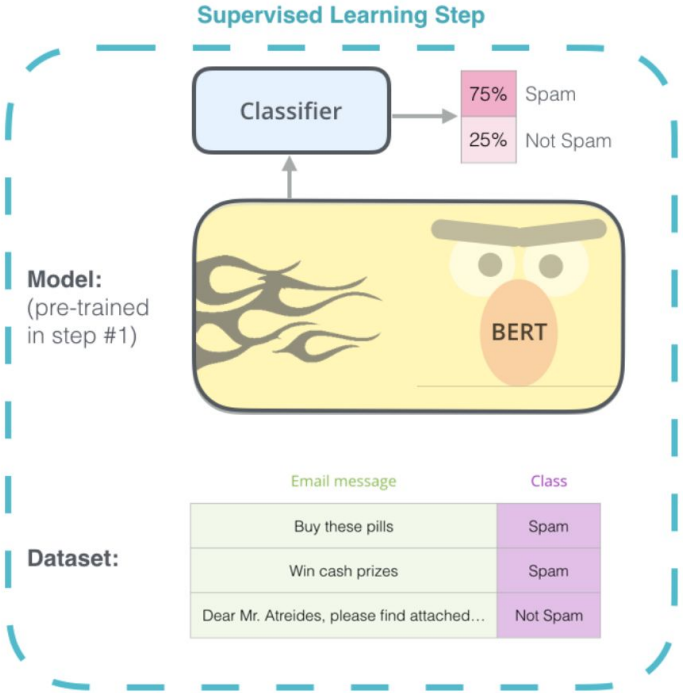
Transfer Learning

1 - Semi-supervised training on large amounts of text (books, wikipedia..etc).

The model is trained on a certain task that enables it to grasp patterns in language. By the end of the training process, BERT has language-processing abilities capable of empowering many models we later need to build and train in a supervised way.



2 - Supervised training on a specific task with a labeled dataset.



Transformers for Computer Vision

ViT

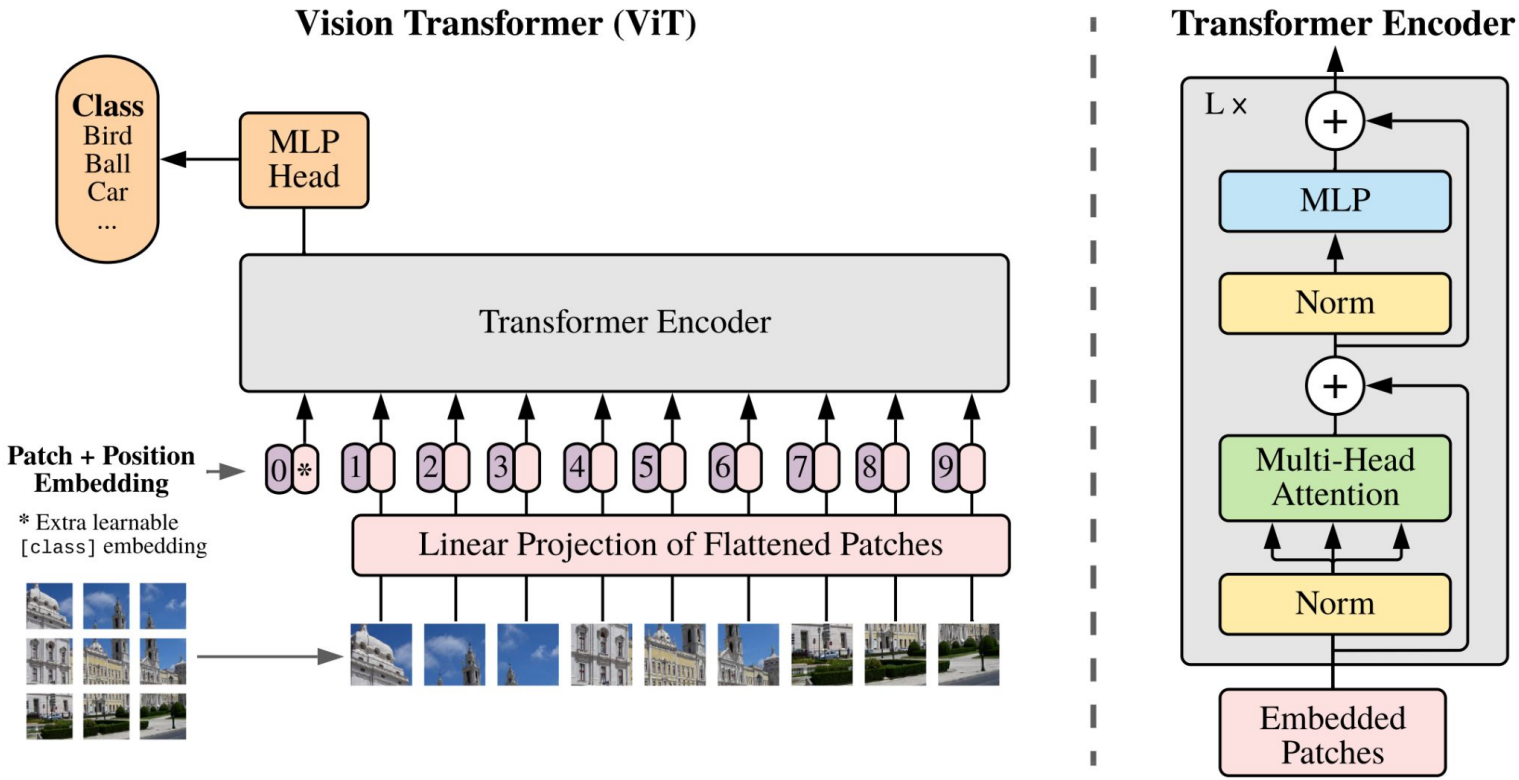
Vision Transformers (ViT) are starting to dominate computer vision.

Compared to CNNs, they achieve **higher accuracies on large datasets** due to their:

- Higher modeling capacity
- Lower inductive biases
- Global receptive fields

CNNs are still on-par or better than ViTs on ImageNet in terms of model complexity or size versus accuracy

ViT


Vision Transformer (ViT)
Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0*
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches
Transformer Encoder
L x
MLP
Norm
Multi-Head Attention
Norm
Embedded Patches

PYTORCH-TRANSFORMERS

https://pytorch.org/hub/huggingface_pytorch-transformers/

Questions?