Transformers

Thursday, 6 January 2022 5:25 PM

References:

Slide :
https://docs.google.com/presentation/d/1ZXFIhYczos679r70Yu8vV9uO6B1J0ztzeDxbnBxD1

Must:
https://lena-voita.github.io/nlp_course/transfer_learning.html

Fine-Tune BERT Model:
https://github.com/PradipNichite/Youtube-Tutorials/blob/main/FineTune_BERT_Model_Yo
https://blog.futuresmart.ai/fine-tuning-hugging-face-transformers-model

NLP code base:
https://index.quantumstat.com

https://towardsdatascience.com/illustrated-guide-to-transformer-cf6969ffa067?gi=be6adeb
https://github.com/aladdinpersson/Machine-Learning-Collection ==best repo.

https://amatriain.net/blog/transformer-models-an-introduction-and-catalog-2d1e9039f376

https://www.youtube.com/watch?v=dichIcUZfOw

https://thegradient.pub/transformers-are-graph-neural-networks/ ==try this new totally

Language Model Comparison:
https://tungmphung.com/a-review-of-pre-trained-language-models-from-bert-roberta-to-e

https://txt.cohere.ai
https://txt.cohere.ai/what-are-transformer-models/

GPT:

S0/mobilepresent?slide=id.g13dd67c5ab8_0_3799

utube.ipynb

bccb99 ==new

/

lectra-deberta-bigbird-and-more/

GPT:
https://jaykmody.com/blog/gpt-from-scratch/
https://jalammar.github.io/how-gpt3-works-visualizations-animations/

Aspect based Sentiment classification:

https://blog.futuresmart.ai/aspect-based-sentiment-analysis-with-pyabsa-hugging-face

Topics:
https://magazine.sebastianraschka.com/p/understanding-encoder-and-decoder

https://lightning.ai/pages/courses/deep-learning-fundamentals/unit-1/1-6-implementing-a-

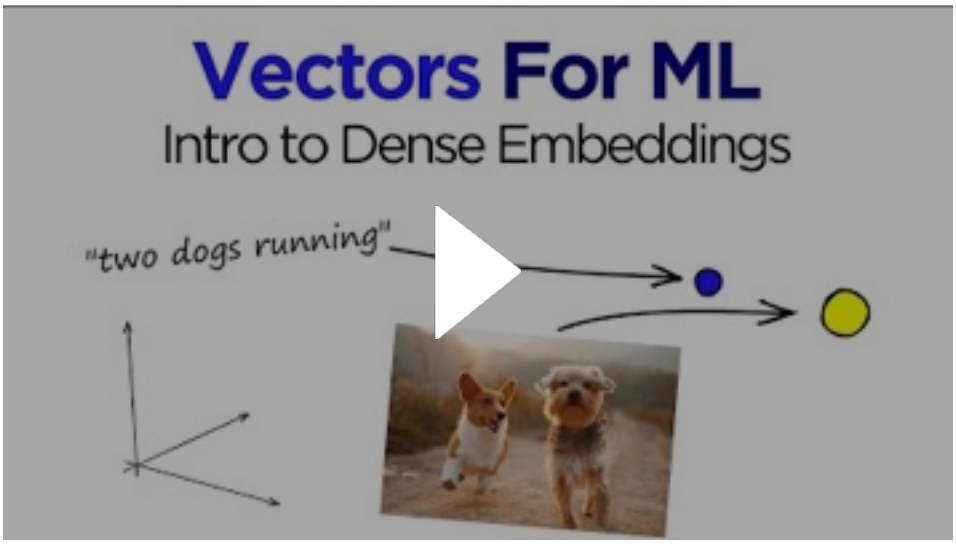
https://learning.edx.org/course/course-v1:Databricks+LLM101x+2T2023/block-v1:Databrick
+type@sequential+block@dc9338e56abd4138b73d0a71cf16c7e8/block-v1:Databricks+LLM

Semantic Search:

References:

0)
Best course:
https://www.pinecone.io/learn/series/nlp/
https://www.pinecone.io/learn/series/nlp/

NLP for Semantic Search Course



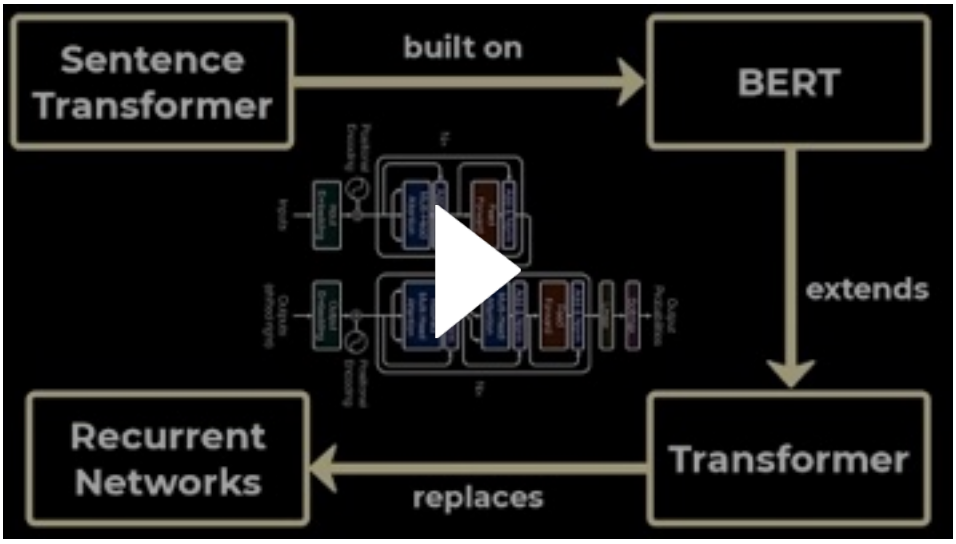
Explained:

-perceptron-in-python-parts-1-3/

ks+LLM101x+2T2023
101x+2T2023+type@vertical+block@20291c0c78534343a993c3f23b1ff350

https://www.pinecone.io/learn/series/nlp/fine-tune-sentence-transformers-mnr/

1)
Sentence Transformers - EXPLAINED!



2)
SetFit and CrossEncoder :

1. **Define your threshold**: Decide on a threshold score that determines the boundary between the classes you're trying to label (e.g., in a binary classification task, you might choose 0.5 as a threshold where scores above 0.5 indicate class 1 and scores below 0.5 indicate class 0).
2. **Prepare your data**: Organize your data into pairs of sentences that you want to compare. This will depend on your specific task.
3. **Use the cross-encoder to score sentence pairs**: Pass each pair of sentences through the cross-encoder to get a similarity score.
4. **Assign labels based on the scores**: Based on the scores and your pre-defined threshold, assign a label to each sentence pair.

Here is an example in pseudocode:

python Copy code

```
# Assume `cross_encoder` is your pre-trained cross-encoder model
# Assume `sentence_pairs` is a list of tuples, where each tuple contains two

threshold = 0.5  # Define your threshold
labels = []

for sentence_pair in sentence_pairs:
    # Use the cross-encoder to score the sentence pair
    score = cross_encoder.predict(sentence_pair)

    # Assign a label based on the score
    if score >= threshold:
```

```
        label = 1  # Class 1 for scores above the threshold
    else:
        label = 0  # Class 0 for scores below the threshold

    labels.append(label)

# Now `labels` contains the assigned labels for each sentence pair in your
```

Regenerate

3)
Set fit and code :
https://huggingface.co/blog/setfit

https://github.com/huggingface/setfit

4)

Cross-Encoders:

- **Input**: Pairs of sentences.
- **Output**: A single score that represents the relationship between the two sentences in each pair, typically a similarity or relevance score.
- **Usage**: Best used when you have a relatively small, fixed number of sentence pairs to evaluate. Common applications include ranking, where you might want to compare a query sentence against a set of possible matches to find the best one.
- **Drawbacks**: Not suitable for large-scale comparisons because the computation grows quadratically with the number of sentences to compare.

Bi-Encoders:

- **Input**: Individual sentences.
- **Output**: A fixed-size vector (embedding) for each input sentence.
- **Usage**: Best used when you need to perform many comparisons or searches, such as semantic search, information retrieval, or clustering. Once you have the embeddings, you can efficiently compare them using cosine similarity or another distance metric.
- **Advantages**: Because you compute the embedding once per sentence, it's much faster for large-scale applications. Comparing new sentences to a pre-computed set of embeddings is very efficient.

random topics:

https://towardsdatascience.com/sentence-transformer-fine-tuning-setfit-outperforms-gpt-3

https://python-bloggers.com/2023/02/few-shot-learning-with-setfit/

Embedding:

3-on-few-shot-text-classification-while-d9a3788f0b4e

https://cla2019.github.io/embedmatrix.pdf

https://nn.labml.ai/transformers/mha.html

Self attention from Scratch :
https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html

5)

Bert Op:::
https://github.com/jamescalam/transformers/tree/main/course
https://tungmphung.com/the-transformer-neural-network-architecture/
1)
Building MLM Training Input Pipeline - Transformers From Scratch #3

The output of BERT consists of two vectors for each input token: the **token representation a**
The **token representation(**batch_size, sequence_length, hidden_size) vector represents the
the **pooled output(**batch_size, hidden_size) represents the overall meaning of the entire se

- https://www.kaggle.com/code/rhtsingh/utilizing-transformer-representations-efficiently/nc
- **pooler output** (batch size, hidden size) - Last layer hidden-state of the first token of the sequenc
- **last hidden state** (batch size, seq Len, hidden size) which is the sequence of hidden states at the

nd the pooled output.
contextual information of the token within the sentence, while
ntence.

otebook
e
output of the last layer

- **hidden states** (n layers, batch size, seq Len, hidden size) - Hidden states for all layers and for all i

```
# Input text
text = "This is a sample sentence."

# Tokenize input text
tokens = tokenizer.tokenize(text)

# Convert tokens to IDs
input_ids = tokenizer.convert_tokens_to_ids(tokens)

# Convert input_ids to tensor
input_tensor = torch.tensor([input_ids])

# Generate contextualized embeddings for input tokens
with torch.no_grad():
    output = model(input_tensor)

# Extract embeddings for each token in the input text
embeddings = output[0][0]

# Compute mean pooling
mean_pooled = torch.mean(embeddings, dim=0)

# Print mean pooled vector
print(mean_pooled)
```

> In the above code, `output` is a tuple that contains the outputs of the BERT model. The first element of `output` is the last hidden state of the model, which is a tensor of shape `[batch_size, sequence_length, hidden_size]`.
>
> The last hidden state is a sequence of contextualized embeddings for each token in the input text. Each contextualized embedding captures the meaning of the corresponding token in the context of the entire input text. The size of the contextualized embeddings is determined by the `hidden_size` parameter of the BERT model, which is typically 768 or 1024.
>
> In the code example, we extract the last hidden state using `output[0]`, which gives us a tensor of shape `[batch_size, sequence_length, hidden_size]`. We then select the embeddings for the first token in the sequence (i.e., the `[CLS]` token) using `output[0][:, 0, :]`. This gives us a tensor of shape `[batch_size, hidden_size]`, which can be used as a fixed-length representation of the entire input

output of the last layer.
ds.

text.

However, in some cases, we may want to use the entire sequence of contextualized embeddings for downstream tasks such as sequence labeling or text generation. In such cases, we can use the entire last hidden state tensor (`output[0]`) as input to the downstream task.

`output[0][:, 0, :]` and `output[0][0]` both refer to the embedding for the first token in the sequence (i.e., the `[CLS]` token) of the last hidden state outputted by the BERT model. However, they differ in their shape and the information they contain.

`output[0][:, 0, :]` is a tensor of shape `[batch_size, hidden_size]`. It represents the embedding for the `[CLS]` token for all samples in the batch. This means that if we input multiple texts into the BERT model, `output[0][:, 0, :]` would contain the embeddings for the `[CLS]` token of each of those texts. This can be useful if we are processing a batch of texts and want to obtain a fixed-length representation of each text.

On the other hand, `output[0][0]` is a tensor of shape `[hidden_size]`. It represents the embedding for the `[CLS]` token for a single input text. In the code example, we are only processing a single input text, so `output[0][0]` contains the embedding for the `[CLS]` token of that text.

If we are using the output of the BERT model as input to a downstream task such as text classification or sentiment analysis, we can use either `output[0][:, 0, :]` or `output[0][0]` as a fixed-length representation of the input text. If we are processing a batch of texts, we would use `output[0][:, 0, :]`, and if we are processing a single text, we would use `output[0][0]`.

The `pooler_output` tensor is a fixed-size representation of the entire input sequence that is generated by applying a linear transformation to the final hidden state of the `[CLS]` token. This transformation is trained during pre-training to produce a good representation of the input sequence for downstream tasks such as classification or question answering.

```
# generate embeddings
outputs = model(input_ids)
last_hidden_states = outputs[0]  # get last hidden state
pooler_output = outputs[1]  # get pooled output representation
```

```
pooler_output = outputs[1]  # get pooled output representation

# Get final hidden state of [CLS] token
cls_embedding = last_hidden_states[:, 0, :]
```

pooler_output = tanh(W * h_cls + b)

Cls_embedding and h_cls are same.

In the BERT model, the `[CLS]` token is a special token that is inserted at the beginning of each input sequence before tokenization. During training, the BERT model learns to encode useful information about the entire input sequence into the `[CLS]` token's final hidden state.

When we extract the `[CLS]` token's final hidden state from the BERT model's output, we obtain a tensor with a shape of `(batch_size, hidden_size)` where `batch_size` is the number of input sequences in the batch and `hidden_size` is the dimensionality of the BERT model's hidden state. This tensor is often referred to as the `h_cls` tensor or the `[CLS]` token embedding.

Pooler Output

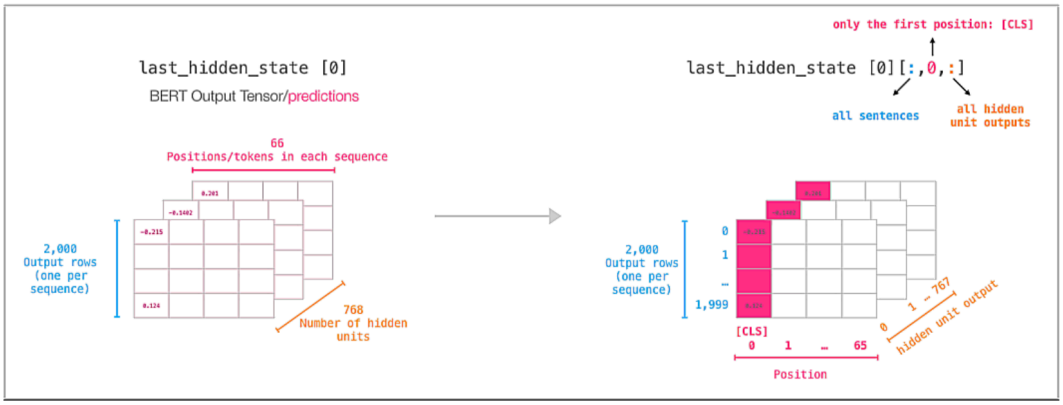
Introduction

This is usually the second output from transformer and the simplest one.

Pooler output is the last layer hidden-state of the first token of the sequence (classification token) further processed by a Linear layer and a Tanh activation function. The Linear layer weights are trained from the next sentence prediction (classification) objective during pretraining.

The `last hidden state` is passed into the pooler and this returns the pooled output. We can deactivate pooler outputs by setting `add pooling layer to False` in model config and passing that to model.

Last Hidden State Output



Introduction

This is the first and default output from models.

Last Hidden State output is the sequence of hidden-states at the output of the last layer of the model. The output is usually `[batch, maxlen, hidden_state]`, it can be narrowed down to `[batch, 1, hidden_state]` for `[CLS]` token, as the `[CLS]` token is 1st token in the sequence. Here, `[batch, 1, hidden_state]` can be equivalently considered as `[batch, hidden_state]`.

Transformer includes two separate mechanisms an encoder and a decoder. BERT has just th
blocks from the transformer.
An encoder that reads the text input and a decoder that produces a prediction for the task.
Since BERT's goal is to generate a language model, only the encoder mechanism is necessary
The GPT2 model discards the encoder part, so there is only one single input sentence rather
inference model.
Apart from that, at inference time BERT generates all its output at once, while GPT is autore

Sentence Transformer : SBERT::
https://www.pinecone.io/learn/sentence-embeddings/

Attention Is All You Need (Transformer) | Paper Explained (best Multi head)

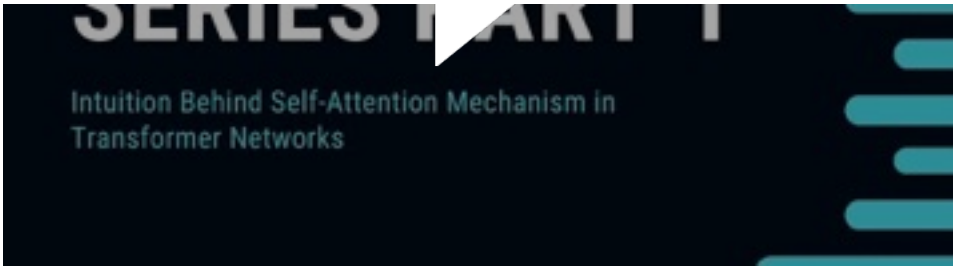
https://www.youtube.com/watch?app=desktop&v=yGTUuEx3GkA

Intuition Behind Self-Attention Mechanism in Transformer Networks ==attention best

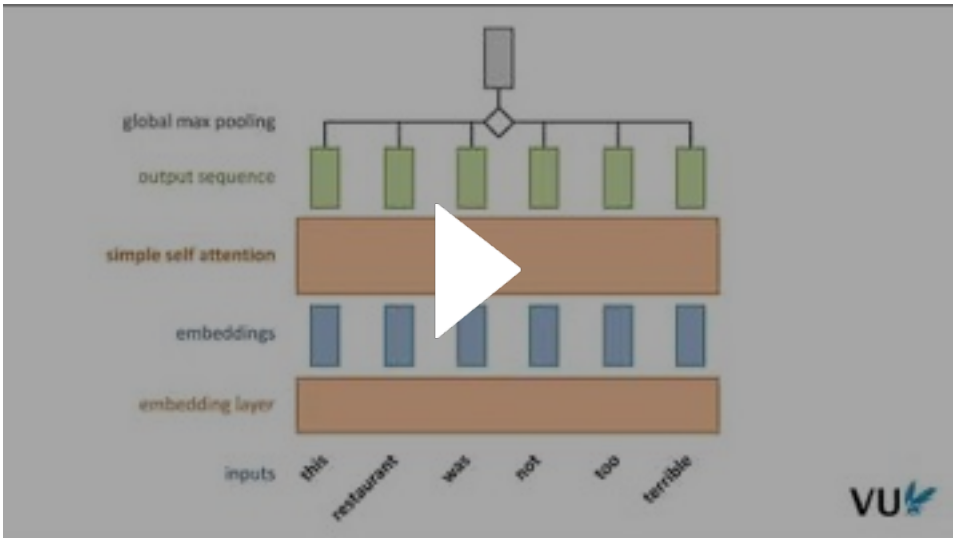
e encoder blocks from the transformer, whilst GPT-2 has just the decoder

y.
than two separate source and target sequences. GPT is an autoregressive

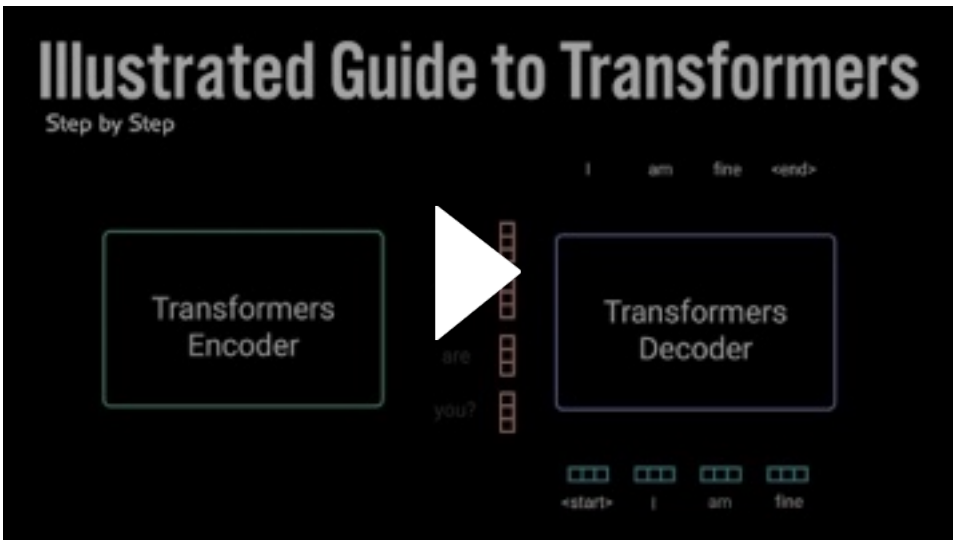
gressive, so it need to iteratively generate one token at a time.



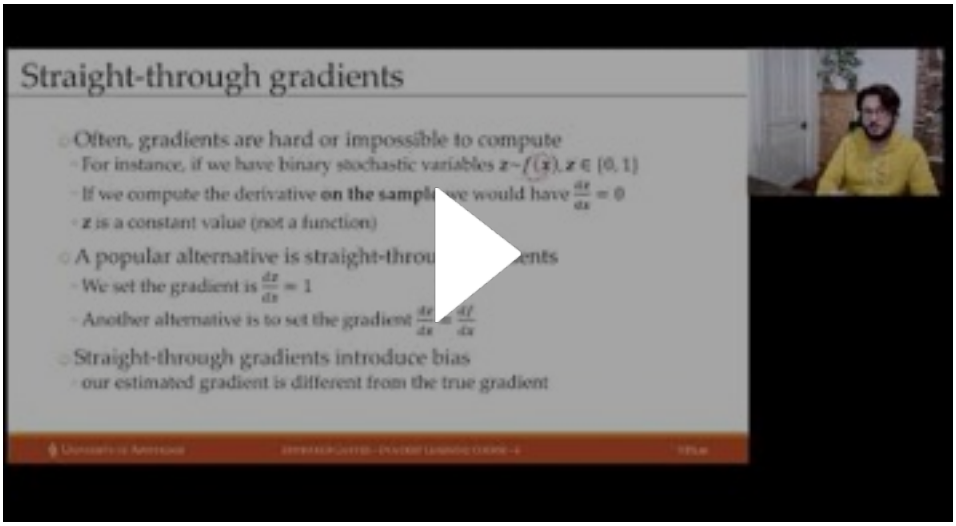
[Lecture 12.1 Self-attention](#) ==best



[Illustrated Guide to Transformers Neural Network: A step by step explanation](#) ==full

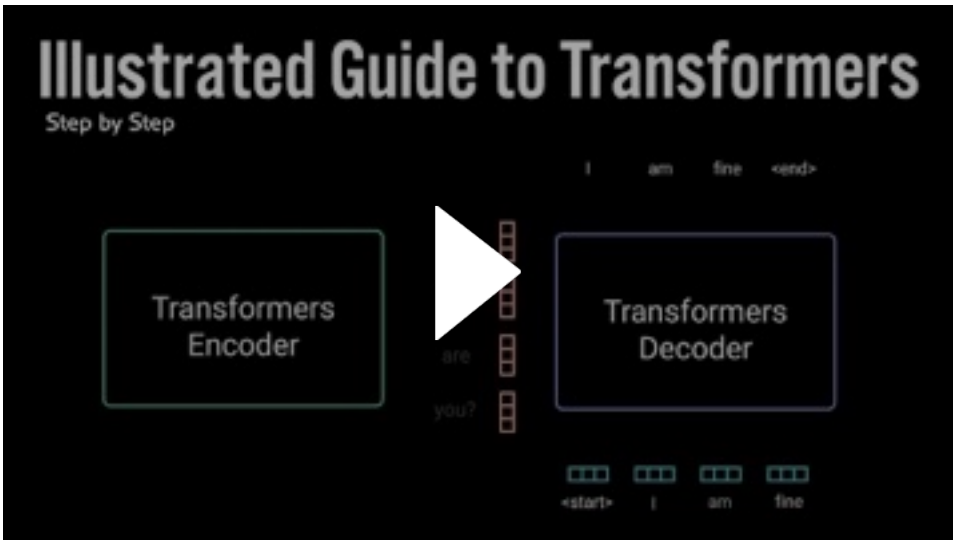


[Lecture 12.4 - Bias and variance in gradients](#) ==latest papers

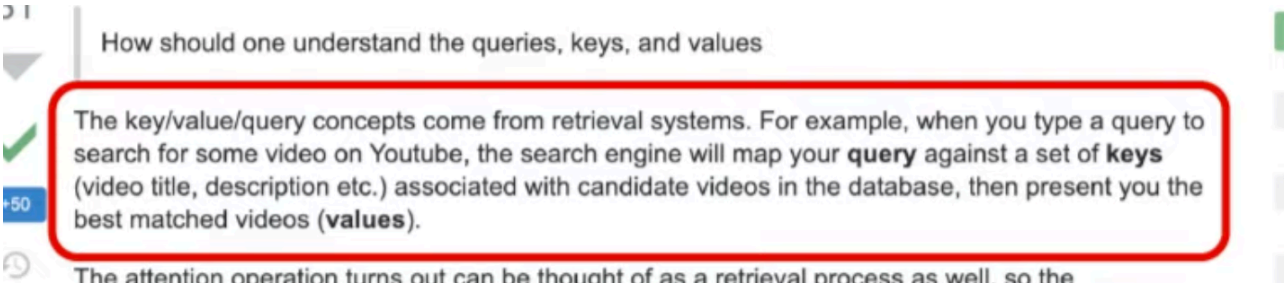


https://nlp.seas.harvard.edu/2018/04/03/attention.html
https://lena-voita.github.io/nlp_course.html

Illustrated Guide to Transformers Neural Network: A step by step explanation



chrom

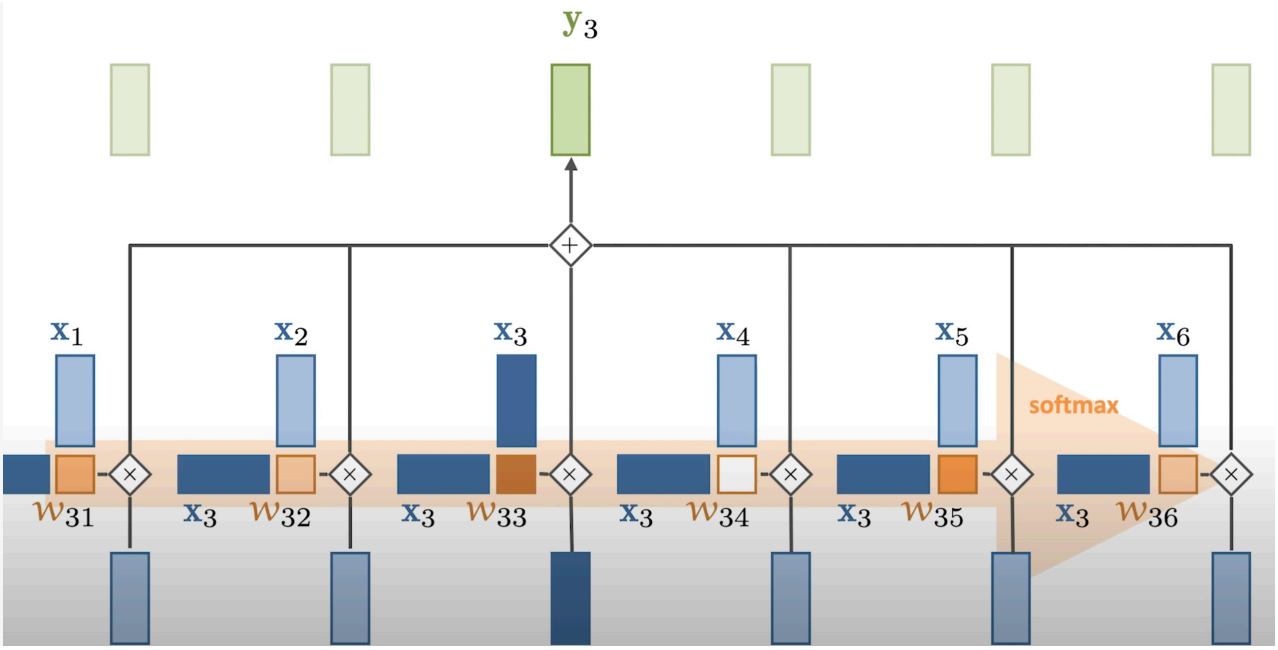


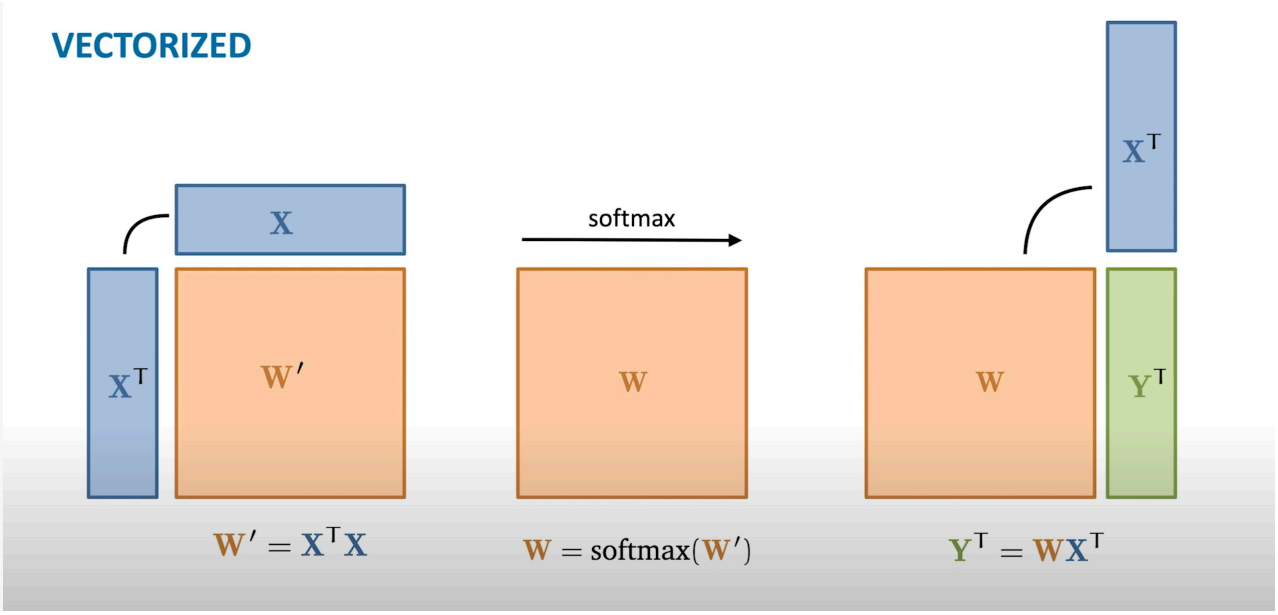
Doubts::

1) Why no parallel ip we can feed @lstm/rnn and why we can in Transformer?
2) Batch Normalization vs Layer Normalization?(wrt NLP)?
3) USE Custom tokenizer in BERT/Transformer based model?
4) https://medium.com/the-dl/transformers-from-scratch-in-pytorch-8777e346ca51 ==c

https://www.novartis.com/about/strategy/data-and-digital/artificial-intelligence/novartis-a

1)Self Attention:





code

i-life-residency-program

TAKE NOTE

In *simple* self-attention w_{ii} (x_i to y_i) usually has the most weight

not a big problem, but we'll allow this to change later.

Simple self-attention has *no parameters*.

Whatever parameterized mechanism generates x_i (like an embedding layer) drives the self attention.

There is a linear operation between X and Y.

non-vanishing gradients through Y = WX^T, vanishing gradients through W = softmax(X^TX).

TAKE NOTE

No problem looking far back into the sequence.

In fact, every input has the same distance to every output.

More of a *set model* than a *sequence model*. No access to the sequential information.

We'll fix by encoding the sequential structure into the embeddings. Details later.

Permutation equivariant.

for any permutation p of the input: p(sa(X)) = sa(p(X))

Scaled Self Attention::: ==> why we do scaling its because it will normalize weight matrix an vanishing gradient issue on SoftMax operation.

$$w'_{ij} = \frac{x_i^T x_j}{\sqrt{k}}$$

<- input dimension

ATTENTION AS A SOFT DICTIONARY

d keeps the weight in certain range==> thus avoid neuron saturations==>

```
d = {'a' : 1, 'b' : 2, 'c' : 3}

d['b'] = 3
```

<- query
<- key
<- value

key	value
a	1
b	2
c	3

ATTENTION AS A SOFT DICTIONARY

Attention is a *soft* dictionary

- key, query and value are vectors
- every key matches the query *to some extent*
  as determined by their dot-product
- a *mixture* of all values is returned
  with softmax-normalized dot products as mixture weights

***Self*-attention**

Attention with keys, queries and values from the same set.

KEY, QUERY AND VALUE TRANSFORMATIONS

introduce matrices **K**, **Q**, **V** for linear transforms
and associated biases

$$\mathbf{k}_i = \mathbf{K}\mathbf{x}_i + \mathbf{b}_k$$
$$\mathbf{q}_i = \mathbf{Q}\mathbf{x}_i + \mathbf{b}_q$$
$$\mathbf{v}_i = \mathbf{V}\mathbf{x}_i + \mathbf{b}_v$$