Νευρο-Ασαφής Υπολογιστική
Neuro-Fuzzy Computing

Διδάσκων –
Δημήτριος Κατσαρός

@ Τμ. ΗΜΜΥ
Πανεπιστήμιο Θεσσαλίας

Διάλεξη 24η: Transformers-I

Transformers

[Mathematical background]

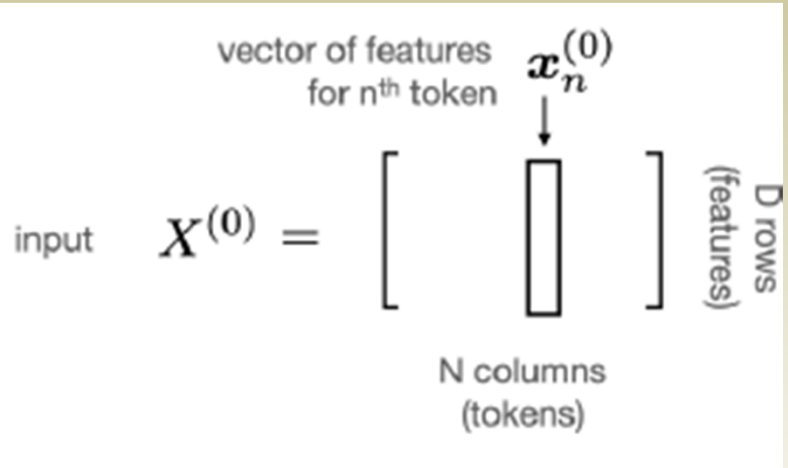
What is a transformer

- The transformer is a neural network component that can be used to learn useful representations of sequences or sets of data-points
- The transformer has driven recent advances in natural language processing, computer vision, and spatio-temporal modeling
- In this lecture we aim for a mathematically precise, intuitive, and clean description of the transformer architecture
  - We will not discuss training as this is rather standard

Preliminaries

Input data format: sets or sequences of tokens

- In order to apply a transformer, data must be converted into a set or sequence of N tokens $x^{(0)}_n$ of dimension D

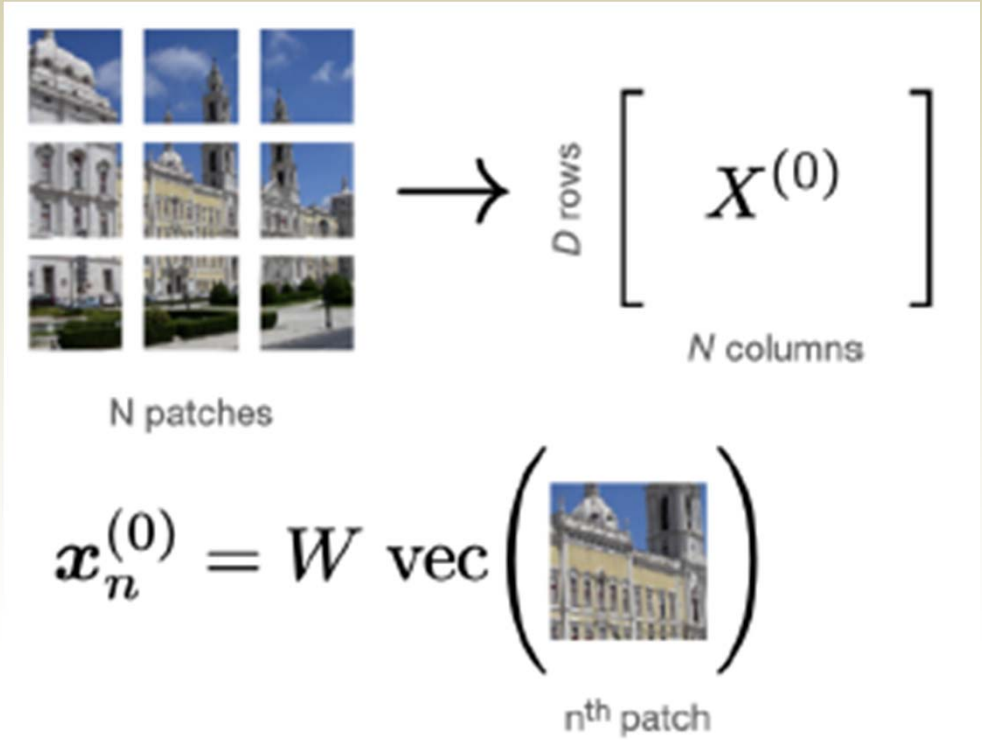


The collection of tokens does not need to have an order and the transformer can handle them as a set (where order does not matter), rather than a sequence

- The tokens can be collected into a matrix $X^{(0)}$ which is D×N
- To give two concrete examples:
  - A passage of text can be broken up into a sequence of words or sub-words, with each word being represented by a single unique vector
  - An image can be broken up into a set of patches and each patch can be mapped into a vector

Preliminaries

- The embeddings can be fixed or they can be learned with the rest of the parameters of the model
  - e.g., the vectors representing words can be optimized or a learned linear transform can be used to embed image patches



Encoding an image
An image is split into N patches.
Each patch is reshaped into a vector by the ***vec*** operator
This vector is acted upon by a matrix W which maps the patch to a D dimensional vector $x^{(0)}_n$
These vectors are collected together into the input X(0)
The matrix W can be learned with the rest of the transformer's parameters

Preliminaries

- A sequence of tokens is a generic representation to use as an input
- Many different types of data can be “tokenised” and transformers are then immediately applicable rather than requiring a bespoke architectures for each modality as was previously the case
  - CNNs for images
  - RNNs for sequences,
- Moreover, this means that you don’t need bespoke handcrafted architectures for mixing data of different modalities — you can just throw them all into a big set of tokens

Transformer's goal

- The transformer will ingest the input data $X^{(0)}$ and return a representation of the sequence in terms of another matrix $X^{(M)}$ which is also of size $D \times N$
- The slice $x_n = X^{(M)}_{:,n}$ will be a vector of features representing the sequence at the location of token n
- These representations can be used for:
  - auto-regressive prediction of the next (n+1)th token
  - global classification of the entire sequence (by pooling across the whole representation)
  - sequence-to-sequence or image-to-image prediction problems, etc
- M denotes the number of layers in the transformer

The transformer block

- The representation of the input sequence will be produced by iteratively applying a transformer block

$$X^{(m)} = \text{transformer-block}(X^{(m-1)})$$

- The block itself comprises two stages:
  - One operating across the sequence
    - This first stage refines each feature independently according to relationships between tokens across the sequence e.g., how much a word in a sequence at position n depends on previous words at position n′, or how much two different patches from an image are related to one another
    - This stage acts horizontally across rows of $X^{(m-1)}$
  - One operating across the features
    - This second stage refines the features representing each token. This stage acts vertically across a column of $X^{(m-1)}$. By repeatedly applying the transformer block the representation at token n and feature d can be shaped by information at token n′ and feature d′

Stage 1: self-attention across the seq

- The output of the first stage of the transformer block is another $D \times N$ array $Y^{(m)}$
- The output is produced by aggregating information across the sequence independently for each feature using an operation called *attention*
- **Attention**. Specifically, the output vector at location n, denoted $y^{(m)}_n$, is produced by a simple weighted average of the input features at location $n' = 1 \ldots N$, denoted $x^{(m-1)}_{n'}$

$$y_n^{(m)} = \sum_{n'=1}^{N} x_{n'}^{(m-1)} A_{n',n}^{(m)}$$

The need for transformers to store and compute N×N attention arrays can be a major computational bottleneck, which makes processing of long sequences challenging

- The weighting is given by a so-called *attention matrix* $A^{(m)}_{n',n}$ which is of size $N \times N$ and normalises over its columns

Stage 1: self-attention across the seq

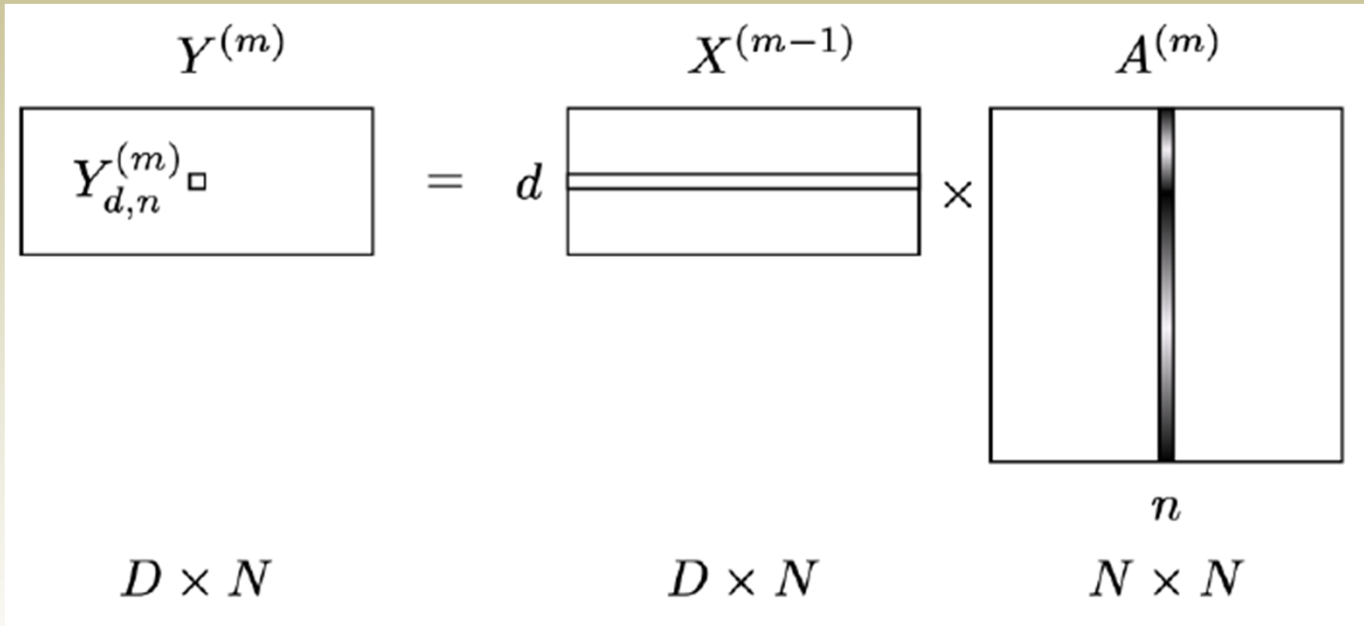
- Intuitively speaking $A^{(m)}_{n',n}$ will take a high value for locations in the sequence n' which are of high relevance for location n
- For irrelevant locations, it will take the value
- For example, all patches of a visual scene coming from a single object might have high corresponding attention values

- **Relationship to Convolutional Neural Networks (CNNs)**. The attention mechanism can recover convolutional filtering as a special case e.g. if $x^{(0)}_n$ is a 1D regularly sampled time-series and $A^{(m)}_{n',n} = A^{(m)}_{n'-n}$ then the attention mechanism in previous slide's Equation becomes a convolution
  - Unlike normal CNNs, these filters have full temporal support. Later we will see that the filters themselves dynamically depend on the input, another difference from standard CNNs
  - We will also see a similarity: transformers will use multiple attention maps in each layer in the same way that CNNs use multiple filters (though typically transformers have fewer attention maps than CNNs have channels)

Stage 1: self-attention across the seq

- We can write the relationship as a matrix multiplication

$$Y^{(m)} = X^{(m-1)}A^{(m)} \qquad \text{(Eq(2))}$$



The shading in the attention matrix represent the elements with a high value in white and those with a low value, near to 0, in black

When training transformers to perform autoregressive prediction, e.g. predicting the next word in a sequence based on the previous ones, a clever modification to the model can be used to accelerate training and inference. This involves applying the transformer to the whole sequence, and using masking in the attention mechanism ($A^{(m)}$ becomes an upper triangular matrix) to prevent future tokens affecting the representation at earlier tokens. Causal predictions can then be made for the entire sequence in one forward pass through the transformer

Stage 1: self-attention across the seq

- **Self-attention**. Where does the attention matrix come from?
- The neat idea in the first stage of the transformer is that the attention matrix is generated from the input sequence itself – so-called *self-attention*
- A simple way of generating the attention matrix from the input would be to measure the similarity between two locations by the dot product between the features at those two locations and then use a softmax function to handle the normalisation i.e.,

$$A_{n,n'} = \frac{e^{x_n^T x_{n'}}}{\sum_{n''=1}^{N} e^{x_{n''}^T x_{n'}}}$$

Stage 1: self-attention across the seq

- However, this naïve approach entangles information about the similarity between locations in the sequence with the content of the sequence itself
- An alternative is to perform the same operation on a linear transformation of the sequence, Ux_n, so that:

$$A_{n,n'} = \frac{e^{x_n^T U^T U x_{n'}}}{\sum_{n''=1}^{N} e^{x_{n''}^T U^T U x_{n'}}}$$

Often you will see attention parameterised as

$$A_{n,n'} = \frac{e^{x_n^T U^T U x_{n'}/\sqrt{D}}}{\sum_{n''=1}^{N} e^{x_{n''}^T U^T U x_{n'}/\sqrt{D}}}$$

Dividing the exponents by the square-root of the dimensionality of the projected vector helps numerical stability, but here, we absorb this term into U to improve clarity

- Typically, U will project to a lower dimensional space i.e., U is KxD dimensional with K < D. In this way only some of the features in the input sequence need be used to compute the similarity, the others being projected out, thereby decoupling the attention computation from the content. However, the numerator in this construction is symmetric. This could be a disadvantage.

Stage 1: self-attention across the seq

- Fortunately, it is simple to generalize the previous attention mechanism to be asymmetric by applying two different linear transformations to the original sequence

$$A_{n,n'} = \frac{e^{x_n^T U_k^T U_q x_{n'}}}{\sum_{n''=1}^{N} e^{x_{n''}^T U_k^T U_q x_{n'}}} \qquad \text{(Eq(3))}$$

- The two quantities that are dot-producted together here $q_n = U_q x_n$ and $k_n = U_k x_n$ are typically known as the *queries* and the *keys*, respectively
- Together Equations 2 and 3 define the self-attention mechanism
- Notice that the $K \times D$ matrices U_q and U_k are the only parameters of this mechanism

Stage 1: self-attention across the seq

- **Multi-head self-attention (MHSA)**. In the self-attention mechanisms described earlier, there is one attention matrix which describes the similarity of two locations within the sequence
- This can act as a bottleneck in the architecture – it would be useful for pairs of points to be similar in some 'dimensions' and different in others

- If attention matrices are viewed as a data-driven version of filters in a CNN, then the need for more filters / channels is clear
- Typical choices for the number of heads H is 8 or 16, lower than typical numbers of channels in a CNN

Stage 1: self-attention across the seq

- In order to increase capacity of the first self-attention stage, the transformer block applies H sets of self-attention in parallel (termed H heads) and then linearly projects the results down to the D × N array required for further processing
- This slight generalization is called *multi-head self-attention*

- The computational cost of multi-head self-attention is usually dominated by the matrix multiplication involving the attention matrix and is therefore $O(H \times D \times N^2)$

Stage 1: self-attention across the seq

$$Y^{(m)} = MHSE_\theta(X^{(m-1)}) = \sum_{h=1}^{H} V_h^{(m)} X^{(m-1)} A_h^{(m)} \tag{4}$$

$$[A_h^{(m)}]_{n,n'} = \frac{e^{\left(k_{h,n}^{(m)}\right)^T q_{h,n'}^{(m)}}}{\sum_{n''=1}^{N} e^{\left(k_{h,n''}^{(m)}\right)^T q_{h,n'}^{(m)}}} \tag{5}$$

$$q_{h,n}^{(m)} = U_{q,h}^{(m)} x_n^{(m-1)} \quad \text{and} \quad k_{h,n}^{(m)} = U_{k,h}^{(m)} x_n^{(m-1)} \tag{6}$$

- Here the H matrices $V^{(m)}_h$ which are D×D project the H self-attention stages down to the required output dimensionality D

Stage 1: self-attention across the seq

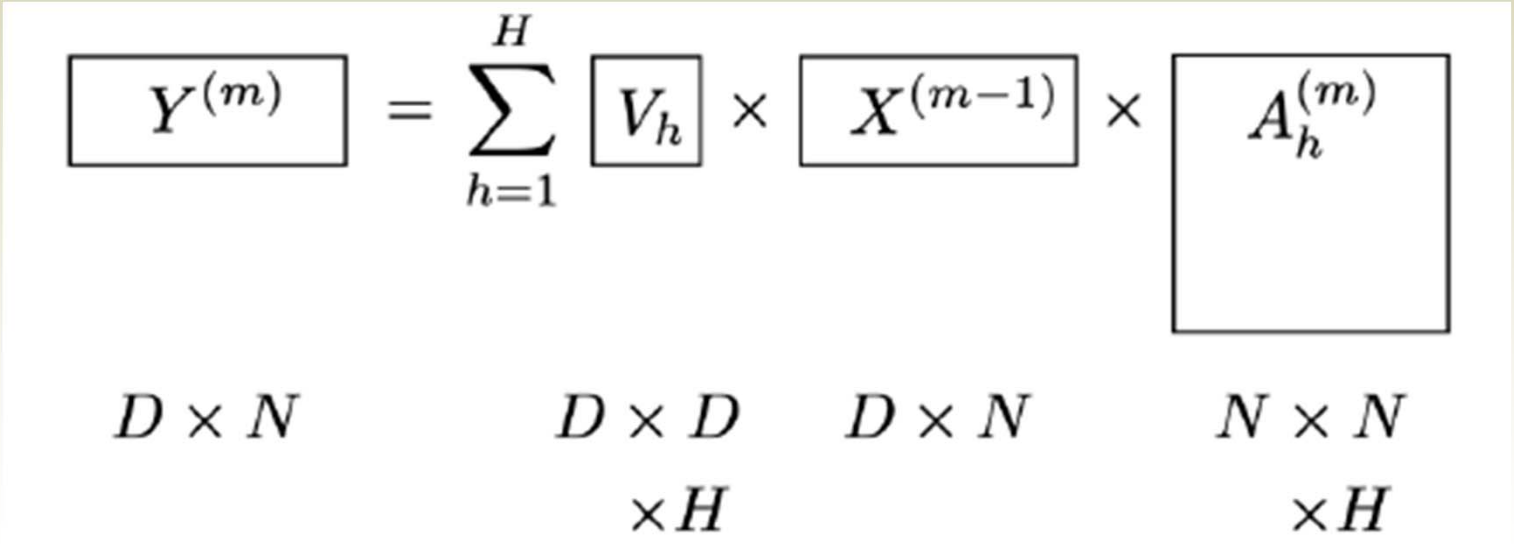
- The product of the matrices $V^{(m)}_h X^{(m-1)}$ is related to the so-called *values* which are normally introduced in descriptions of self-attention along side queries and keys
- In the usual presentation, there is a redundancy between the linear transform used to compute the values and the linear projection at the end of the multi-head self-attention, so we have not explicitly introduced them here
- The standard presentation can be recovered by setting V_h to be a low-rank matrix $V_h = U_h U_{v,h}$ where U_h is DxK and $U_{v,h}$ is KxD
- Typically K is set to K = D/H so that changing the number of heads leads to models with similar numbers of parameters and computational demands

Stage 1: self-attention across the seq

- The addition of the matrices $V^{(m)}_h$, and the fact that retaining just the diagonal elements of the attention matrix $A^{(m)}$ will interact the signal instantaneously with itself, does mean there is some cross-feature processing in multi-head self-attention, as opposed to it containing purely cross-sequence processing
  - However, the stage has limited capacity for this type of processing and it is the job of the second stage to address this

Stage 1: self-attention across the seq

- **MHSA schematically**. Multi-head attention comprises the following parameters $\theta = \{U_{q,h}, U_{k,h}, V_h\}^H_{h=1}$ i.e., 3H matrices of size K × D, K × D, and D × D respectively



Stage-2: MLP across features

- The second stage of processing in the transformer block operates across features, refining the representation using a non-linear transform
- To do this, we simply apply a multi-layer perceptron (MLP) to the vector of features at each location n in the sequence,

$$x_n^{(m)} = MLP_\theta(y_n^{(m)})$$

- Notice that the parameters of the MLP, θ, are the same for each location n

- The MLPs used typically have one or two hidden-layers with dimension equal to the number of features D (or larger)
- The computational cost of this step is therefore roughly N×D×D
- If the feature embedding size approaches the length of the sequence D ≈ N, the MLPs can start to dominate the computational complexity (e.g., this can be the case for vision transformers which embed large patches)

The transformer block

- We can now stack MHSA and MLP layers to produce the transformer block
  - Rather than doing this directly, we make use of two ubiquitous transformations to produce a more stable model that trains easier:
  - **residual connections**, and
  - **normalization**
- **Residual connections**. The use of residual connections is widespread across machine learning as they make initialization simple, have a sensible inductive bias towards simple functions, and stabilize learning
- Instead of directly specifying a function $x^{(m)} = f_\theta(x^{(m-1)})$, the idea is to parameterize it in terms of an identity mapping and a residual term

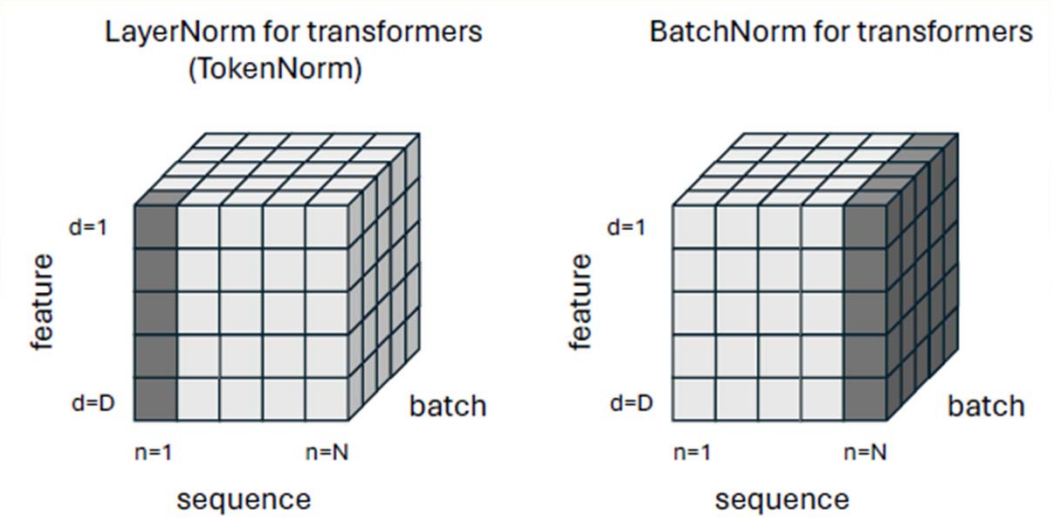
$$x^{(m)} = x^{(m-1)} + res_\theta(x^{(m-1)})$$

The transformer block

- Equivalently, this can be viewed as modeling the differences between the representation $x^{(m)} - x^{(m-1)} = res_{\theta}(x^{(m-1)})$ and will work well when the function that is being modeled is close to identity
- This type of parameterization is used for both the MHSA and MLP stages in the transformer, with the idea that each applies a mild non-linear transformation to the representation
- Over many layers, these mild non-linear transformations compose to form large transformations

The transformer block

- **Token normalization**. Normalization, such as LayerNorm and BatchNorm, is a means to stabilize learning
- There are many potential choices for how to compute normalization statistics (see figure below), but the standard approach is use LayerNorm which normalizes each token separately, removing the mean and dividing by the standard deviation



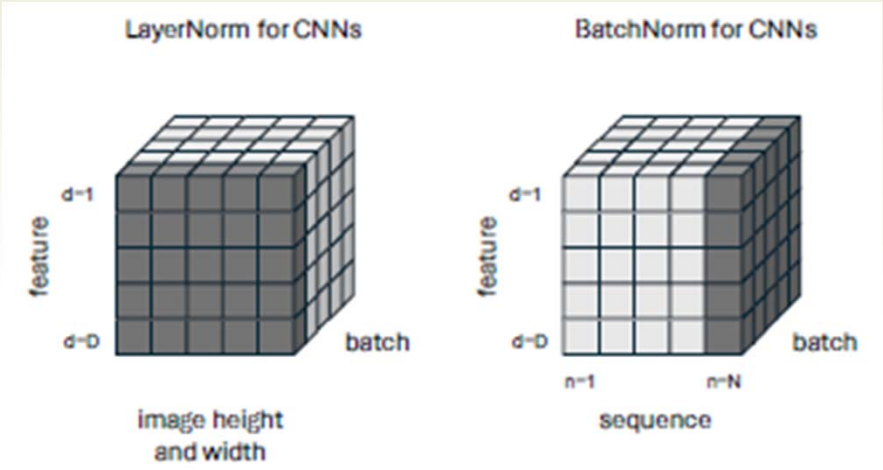
Transformers perform layer normalization which normalizes the mean and standard deviation of each individual token in each sequence in the batch. Batch normalization, which normalizes over the feature and batch dimension together, is found to be far less stable. Other flavors of normalization are possible and potentially under-explored e.g., instance normalization would normalize across the sequence dimension instead.

The transformer block

$$\bar{x}_{d,n} = \frac{1}{\sqrt{var(x_n)}}(x_{d,n} - \text{mean}(x_n))\gamma_d + \beta_d = \text{LayerNorm}(X)_{d,n}$$

$\text{mean}(x_n) = 1/D \sum_{d=1}^{D} x_{d,n}$ and $\text{var}(x_n) = 1/D \sum_{d=1}^{D} (x_{d,n} - \text{mean}(x_n))^2$

- The two parameters γ_d and β_d are a learned scale and shift
- As this transform normalizes each token individually and as LayerNorm is applied differently in CNNs (figure below), let us call it as *TokenNorm*
- This transform stops feature representations blowing up in magnitude as nonlinearities are repeatedly applied through neural networks. In transformers, LayerNorm is usually applied in the residual terms of both the MHSA and MLP stages



In CNNs LayerNorm is conventionally applied to both the features and across the feature maps (i.e. across the height and width of the images). As the height and width dimension in CNNs corresponds to the sequence dimension, 1 . . .N of transformers, the term 'LayerNorm' is arguably used inconsistently (compare to previous' slide figure). I would prefer to call the normalisation used in transformers 'token normalisation' instead to avoid confusion. Batch normalisation is consistently defined

The transformer block

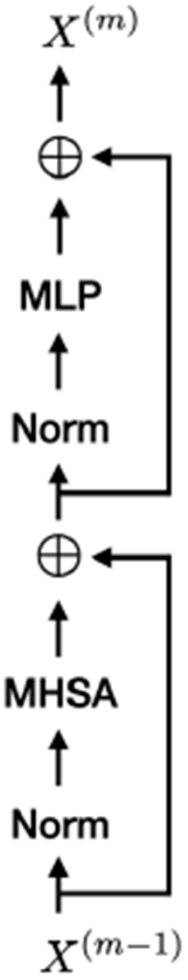
- Putting this all together, we have the standard transformer block shown schematically below

$$X_n^{(m)} = Y_n^{(m)} + \mathrm{MLP}(\bar{Y}_n^{(m)})$$

$$\bar{Y}^{(m)} = \mathrm{LayerNorm}(Y^{(m)})$$

$$Y^{(m)} = X^{(m-1)} + \mathrm{MHSA}(\bar{X}^{(m-1)})$$

$$\bar{X}^{(m-1)} = \mathrm{LayerNorm}(X^{(m-1)})$$



The transformer block.
Residual connections are added to the multihead self-attention (MHSA) stage and the multi-layer perceptron (MLP) stage
Layer normalization is also applied to the inputs of both the MHSA and the MLP. They are then stacked.
This block can then be repeated M times

Positional encoding

- The transformer treats the data as a set — if you permute the columns of $X^{(0)}$ (i.e. re-order the tokens in the input sequence) you permute all the representations throughout the network $X^{(m)}$ in the same way
- This is key for many applications since there may not be a natural way to order the original data into a sequence of tokens
  - For example, there is no single 'correct' order to map image patches into a one dimensional sequence
- This presents a problem since positional information is key in many problems and the transformer has thrown it out
  - The sequence 'herbivores eat plants' should not have the same representation (up to permutation) as 'plants eat herbivores'
  - Nor should an image have the same representation as one comprising the same patches randomly permuted

Positional encoding

- There is a simple fix for this: the location of each token within the original dataset should be included in the token itself, or through the way it is processed
- There are several options how to do this
  - one is to include this information directly into the embedding $X^{(0)}$. E.g. by simply adding the position embedding (surprisingly this works) or concatenating
- The position information
  - can be fixed e.g. adding a vector of sinusoids of different frequencies and phases to encode position of a word in a sentence, OR
  - it can be a free parameter which is learned, as it often done in image transformers
- There are also approaches to include relative distance information between pairs of tokens by modifying the self-attention mechanism

Auto-regressive language modeling

- In auto-regressive language modeling the goal is to predict the next word w_n in the sequence given the previous words $w_{1:n-1}$, that is to return

$$p(w_n=w \mid w_{1:n-1})$$

- Two modifications are required to use the transformer for this task
  - a change to the body to make the architecture efficient and
  - the addition of a head to make the predictions for the next word

Auto-regressive language modeling

- **Modifications to the body**: **Auto-regressive masking**. Applying the version of the transformer we have covered so far to auto-regressive prediction is computationally expensive, both during training and testing
  - To see this, note that AR prediction requires making a sequence of predictions: you start by predicting the first word $p(w_1 = w)$, then you predict the second given the first $p(w_2 = w \mid w_1)$, then the third word given the first two $p(w_3 = w \mid w_1, w_2)$, and so on until you predict the last item in the sequence $p(w_N = w \mid w_{1:N-1})$
  - This requires applying the transformer $N-1$ times with input sequences that grow by one word each time: $w_1, w_{1:2}, \ldots, w_{1:N-1}$. This is very costly at both training-time and test-time

Auto-regressive language modeling

- Fortunately, there is a neat way around this by enabling the transformer to support incremental updates whereby if you add a new token to an existing sequence, you do not change the representation for the old tokens
- To make this property clear, we will define it mathematically:

  Let the output of the incremental transformer applied to the first n words be denoted

  $$X^{(n)} = \text{transformer-incremental}(w_{1:n})$$

  Then, the output of the incremental transformer when applied to n+1 words is

  $$X^{(n+1)} = \text{transformer-incremental}(w_{1:n+1})$$

Auto-regressive language modeling

- In the incremental transformer $X^{(n)} = X^{(n+1)}_{1:D,1:n}$ i.e., the representation of the old tokens has not changed by adding the new one
- If we have this property then:
  1. At test-time auto-regressive generation can use incremental updates to compute the new representation efficiently
  2. At training time we can make the N auto-regressive predictions for the whole sequence $p(w_1 = w)p(w_2 = w \mid w_1)p(w_3 = w \mid w_1, w_2) \ldots p(w_N = w \mid w_{1:N-1})$ in a single forwards pass

Auto-regressive language modeling

- Unfortunately, the standard transformer introduced above does not have this property due to the form of the attention used
  - Every token attends to every other token, so if we add a new token to the sequence then the representation for every token changes throughout the transformer
- However, if we mask the attention matrix so that it is upper-triangular $A_{n,n'} = 0$ when $n > n'$ then the representation of each word *only depends on the previous words*
- This then gives us the incremental property as none of the other operations in the transformer operate across the sequence

Auto-regressive language modeling

- **Adding a head**. We're now almost set to perform auto-regressive language modeling:
- We apply the masked transformer block M times to the input sequence of words
- We then take the representation at token n−1, that is $x^{(M)}_{n-1}$ which captures causal information in the sequence at this point, and generate the probability of the next word through a softmax operation

$$p(w_n = w | w_{1:n-1}) = p(w_n = w | x^{(M)}_{n-1}) = \frac{e^{g_w^T x^{(M)}_{n-1}}}{\sum_{w=1}^{W} e^{g_w^T x^{(M)}_{n-1}}}$$

- Here, W is the vocabulary size, the wth word is w and $\{g_w\}^W_{w=1}$ are softmax weights that will be learned

Concluding remarks

- We have not talked about loss functions or training in any detail, but this is because rather standard deep learning approaches are used for these
- Briefly transformers are typically trained using the Adam optimizer
- They are often slow to train compared to other architectures and typically get more unstable as training progresses
  - Gradient clipping,
  - Decaying learning rate schedules, and
  - Increasing batch sizes through training

  help to mitigate these instabilities, but often they still persist.