Advanced Methods in Text Analytics

UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

Transformers

Why Transformers?

- **Most successful** and **commonly used** deep learning architecture in NLP
  - By far!
- **Resilient technology**
  - RNNs improved over FNNs
  - LSTMs improved over vanilla RNNs
  - Attention improved over vanilla encoder-decoder architectures
  - Transformer proposed in 2017/2018, still largely unchanged
  - Alternative architectures do exist/are proposed, e.g. Mamba, LSTMs
- Main advantages over RNNs (previous state-of-the-art models)
  - **Attention over arbitrarily large inputs**, i.e. no recurrent connections
  - **Flexible/powerful attention** mechanism
  - **Scalable!** Parallelizable, important contrast to RNN-based models

Outline

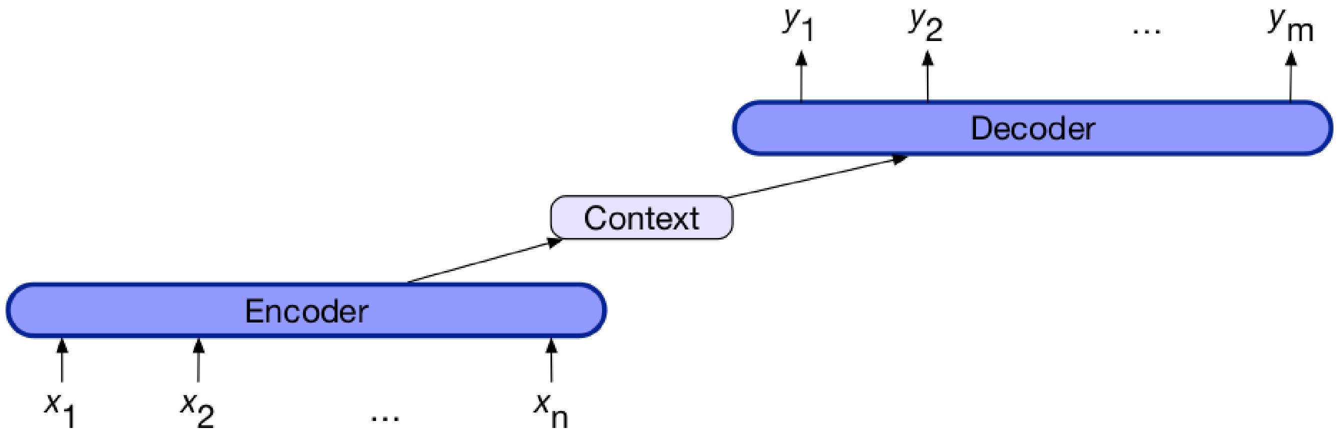
1. Recap: Attention

2. Self-Attention

3. The Transformer Architecture

Recap: Attention

Encoder-Decoder Architecture



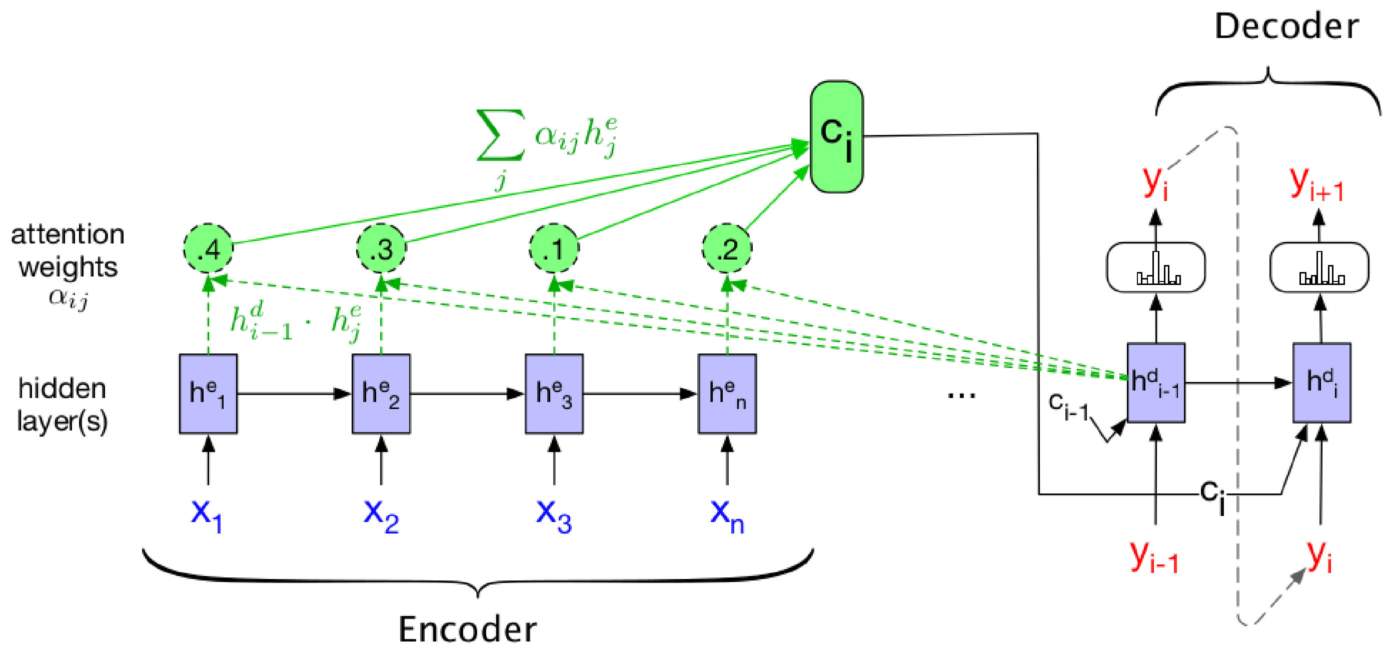
- **Goal:** create contextually appropriate sequence of arbitrary length
  - Known as seq2seq models
- Components
  - **Encoder:** typically, an RNN
  - **Context vector:** produced by encoder (typically last hidden state in RNN)
  - **Decoder:** RNN, produces task-dependent output based on context vector
    - Thus, input sequence represented entirely by context vector
- **Common applications:** machine translation, dialogue systems, etc.

Attention (1)

- Seminal work in machine translation: Bahdanau et al. (2015)
- Recall: RNNs have a hard time using information far back in time
  - Thus, the **context vector may not encode everything we need**
  - LSTMs somewhat address this, but…
- **Why not allow decoder to access input sequence?**
- In encoder-decoder architecture:
  - Decoder accesses input via hidden states of encoder
  - **Context vector** is now **weighted sum of hidden states in encoder**
- **Important:** context vector dependent on decoder state!
  - Thus, model “attends to” different parts of input to produce different parts of output
- Let’s look at all of this in more detail!

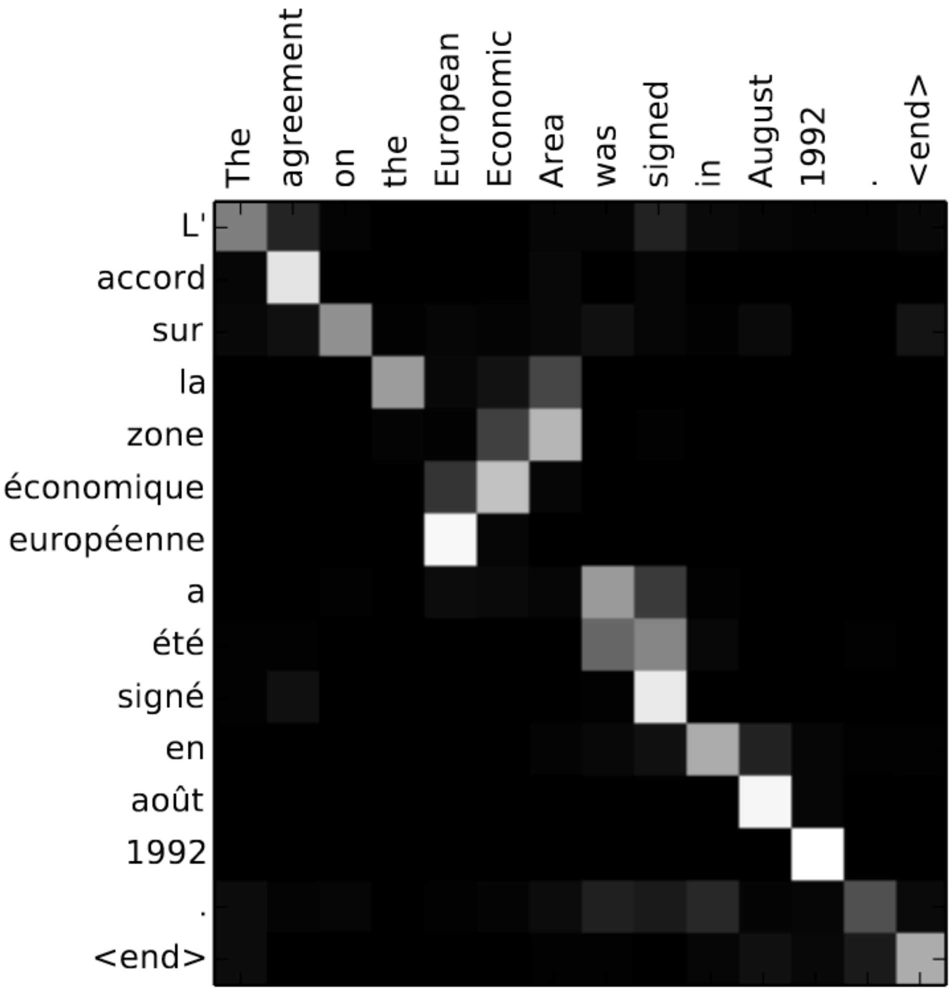
Attention (2)

- c_i is context vector for output i, defined as $c_i = \Sigma_{j=1}^{n} \alpha_{ij} h_j^e$
  - That is, c_i is a weighted sum of encoder hidden states
- Coefficients α_{ij} known as **attention weights**
  - Encode **how much attention is paid to input *j*** to produce output *i*
  - Decoder output h_{i-1}^d necessary, encodes output i - 1 (dashed lines)



Attention (3)

- Entry *i,j* corresponds to attention weight for input *i* given target token *j*
- We can see most words are translated 1-to-1, i.e. look at *n*th input word to produce *n*th output word
- But some are not so simple!
- To produce word *européenne*, the 7th output word, model looks at 5th input word, the actual relevant one!
- **Relevant input can be far!**
  - “Ich habe heute Abend Bratwurst mit Bröt und Kartoffeln *gegessen*.”
  - “Tonight I *ate*…”



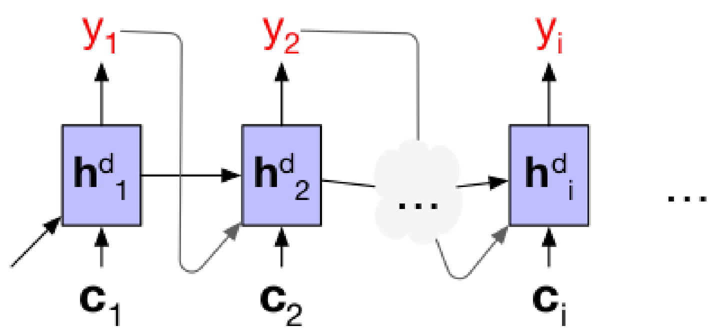
How to Compute Attention Weights?

UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

- Two steps:
  1. Compute **attention scores**
  2. Combine attention scores to produce **attention weights**
- **Attention scores:** how relevant each input is to encoder's last state
  - Each input -> hidden states in encoder $\boldsymbol{h}_j^e$
  - Each output -> decoder hidden states $\boldsymbol{h}_i^d$
- Common approach: **dot-product attention**
  - *score(*output *i,* input *j)* = $(\boldsymbol{h}_i^d)^\top \boldsymbol{h}_j^e$
  - Used to compute output $i + 1$
  - Relevance seen as similarity
- Similarities computed between each input token and last decoder state
  - We still don't know the *relative* relevance across input words
- **Attention weights:** encode *relative* relevance across input words
  - α_{ij} = *softmax(score(output i, input j))*, i.e. mass of α_{ij} w.r.t. all other $\alpha_{ij'}$

Attention: In Short

- To produce output *i+1*:
  1. Compute **attention scores** using last hidden state of decoder's $\boldsymbol{h}_i^d$, e.g. with dot-product attention: *score(*output *i,* input *j)* = $(\boldsymbol{h}_i^d)^\mathsf{T}\, \boldsymbol{h}_j^e$
  2. Compute **attention weights**, e.g. using softmax across all attention scores
  3. Compute **context vector** $\boldsymbol{c}_i$ as linear combination of encoder hidden states, where coefficients are attention weights, i.e. $\boldsymbol{c}_i = \Sigma_j\, \alpha_{ij}\, \boldsymbol{h}_j^e$
  4. Use $\boldsymbol{c}_i$, along with y_i and $\boldsymbol{h}_i^d$, to produce output y_{i+1}



- Thus, attention allows a seq2seq model to see *dynamic* representation of input at each output step
  - Different operations can be used to compute scores and weights

Self-Attention

The Heart of Transformers

- Transformers were introduced by [Vaswani et al.]() in 2017
- Virtually all of state-of-the-art NLP is based on transformers, e.g.
  - **BERT** and other **masked language models** that provide word representations (covered soon)
  - The **GPT family** of text-generating language models (covered soon)
- Deep architecture with many components
- **Key component** of the transformer architecture: **self-attention**
  - They proposed to drop the RNNs and "simply" focus on attention
  - In reality, more components aside from attention
  - But self-attention is **main innovation**, allowed for **flexible and scalable attention over long sequences**
  - More on this later
- First, let's focus on self-attention
  - Then on the overall architecture of the model

Attention as Information Retrieval

- **Useful intuition** about attention **comes from information retrieval**
- Say you open YouTube and input the **query** *"cats dressed as Batman"*
- The search system represents each video with a set of **keys**
  - Think attributes, class properties
  - For example: *title*, *description*, *channel_name*, *publication_date*, etc.
- For any query-key match the system finds, it returns a **value**
  - In this case, values are relevant videos, e.g. those with text similar to your query in the title, description, etc.
- We can see attention as such a query-keys-values system
  - Given attention score *score*(output i, input j) = $(\boldsymbol{h}_i^d)^\top \boldsymbol{h}_j^e$, decoder state $\boldsymbol{h}_i^d$ is a **query**, encoder state $\boldsymbol{h}_j^e$ is a **key**
  - Context vector $\boldsymbol{c}_i$ represents retrieved values, but as linear combination?
  - Yes, so $\boldsymbol{c}_i = \Sigma_j\, \alpha_{ij} \boldsymbol{h}_j^e$ where $\boldsymbol{h}_j^e$ are again used as values
- Thus, attention can be seen as a *soft-retrieval* system
  - It returns relevant values, but each weighted by how relevant they are

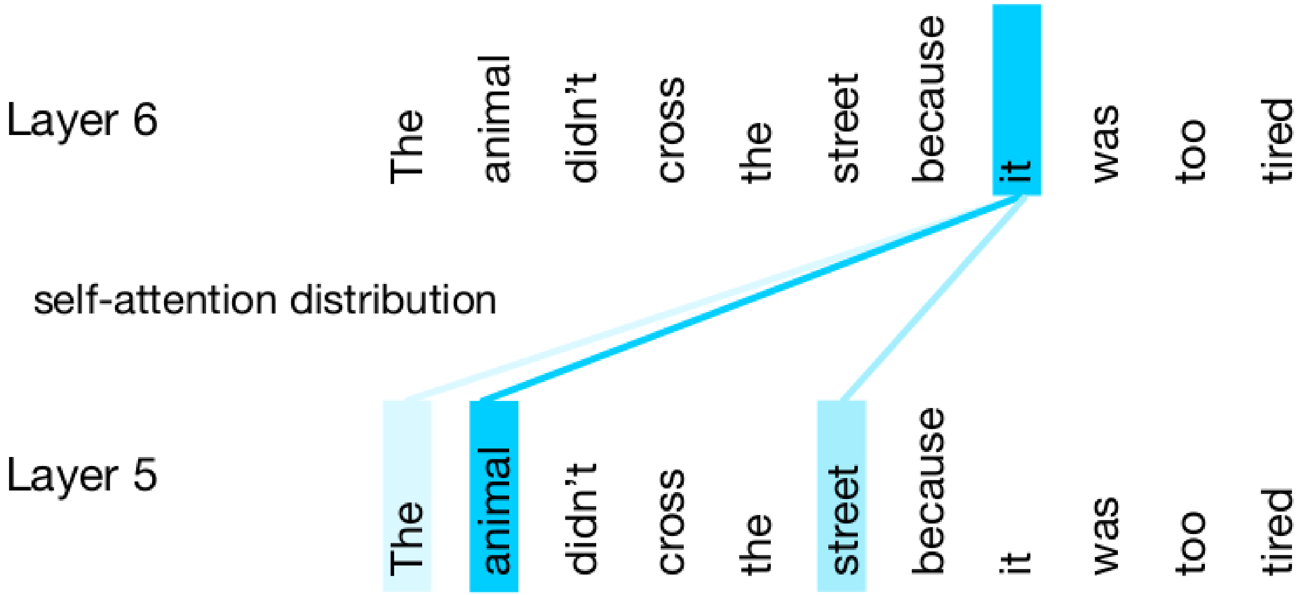
Self-Attention (1)

- **Attention:** compare item of interest to collection of *other items* in a way that reveals their relevance in the current context
  - In encoder-decoder architecture, item of interest is *decoder* state, i.e. output sequence so far
  - Collection of other items is hidden states of *encoder*, i.e. input sequence
- **Self-attention:** compare each token in given sequence, to all other tokens in the *same sequence*, i.e. item of interest is in same collection of items to compare with
- Thus, we use **representations from the same sequence as queries, keys and values**
  - $score(\text{input } i, \text{ input } j) = (\boldsymbol{x}_i^e)^\top \boldsymbol{x}_j^e$
  - $\alpha_{ij} = softmax(score(input\ i, input\ j))$
  - $\boldsymbol{y}_i = \Sigma_j\, \alpha_{ij}\, \boldsymbol{x}_j^e$

  where $\boldsymbol{x}_i$ are representations of given sequence, and $\boldsymbol{y}_i$ are representations after attention, i.e. **contextualized** representations

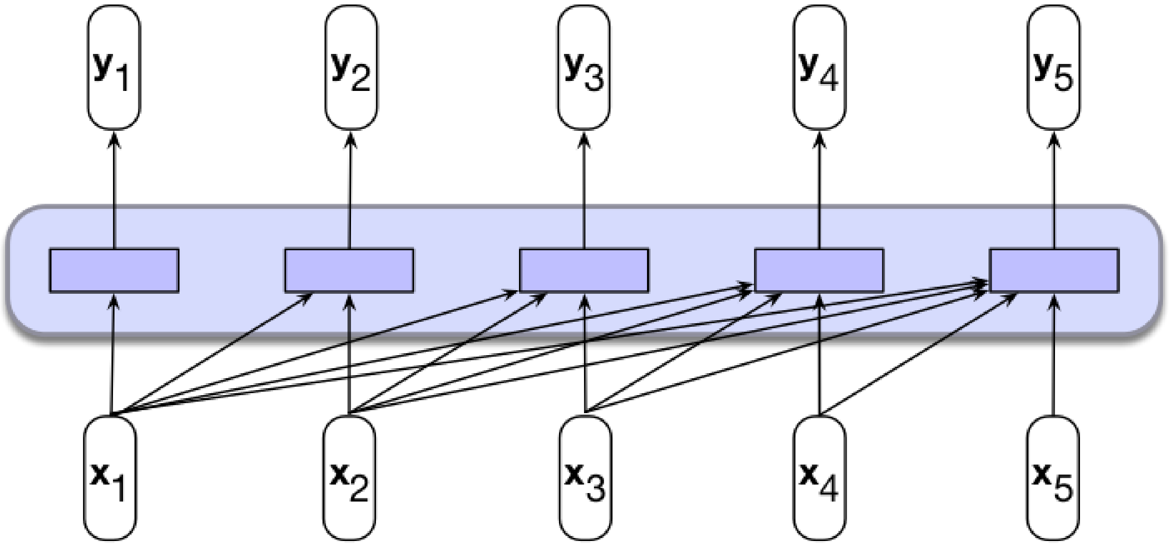
Self-Attention (2)

- **Again**, to produce each output token we compute (potentially) different context vectors
  - Thus, encoders produce different representations at each output step
  - These representations depend on context, i.e. output produced so far, other tokens in input sequence
  - Hence, **contextualized** representations (unlike *static* word embeddings)



Self-Attention Layer (1)

- **Input:** sequence of *n* tokens
- **Output:** sequence of *n* contextualized tokens



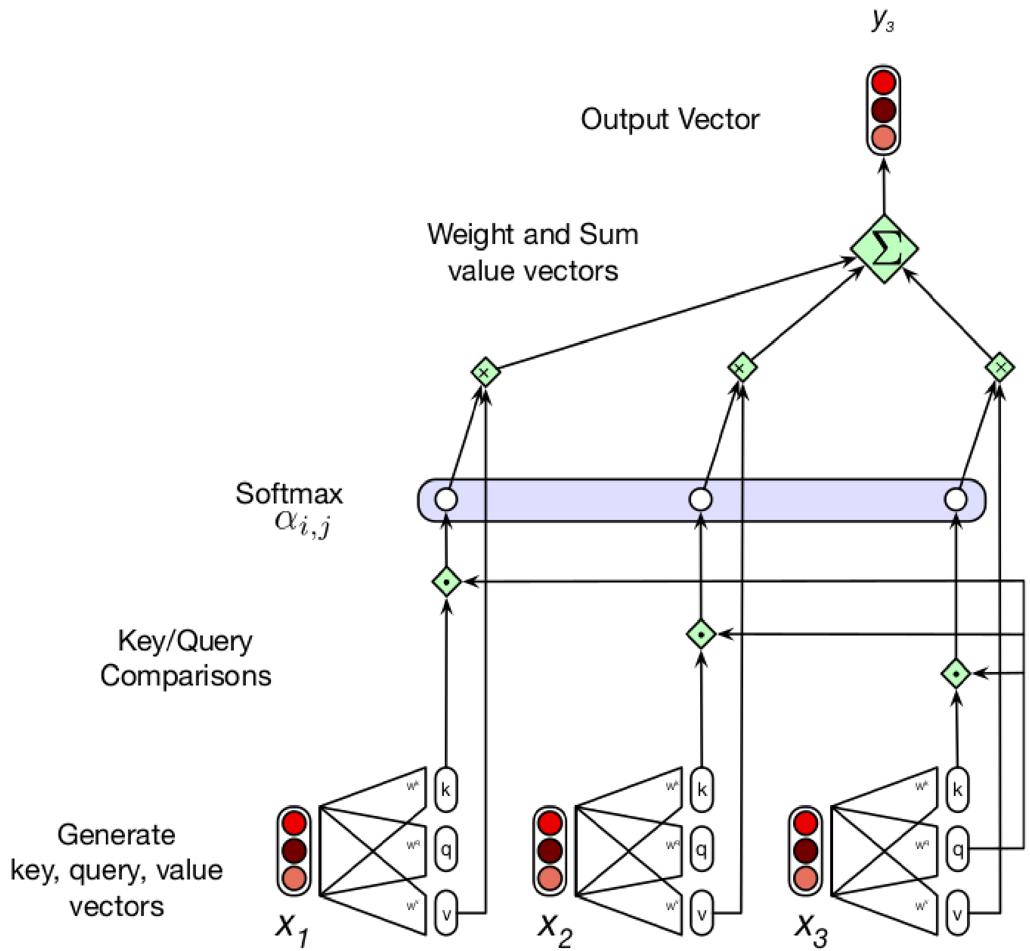
- Layer is **parameterized by** matrices $\boldsymbol{W}^Q$, $\boldsymbol{W}^K$ and $\boldsymbol{W}^V$
  - Each a linear transformation applied to input tokens $\boldsymbol{x}_i$
  - **Each transformation** is applied when tokens are used in the different roles: **queries, keys** and **values**

Self-Attention Layer (2)

- Note the contrast to attention so far, which was not parameterized!
  - Attention scores: dot-product
  - Attention weights: softmax
- Before, model had to learn token representations that could be combined to provide useful context vectors
  - The **parameters in self-attention** allow for a **more flexible attention mechanism**
- Specifically, $\boldsymbol{q}_i = \boldsymbol{W}^Q\boldsymbol{x}_i$, $\boldsymbol{k}_i = \boldsymbol{W}^K\boldsymbol{x}_i$ and $\mathbf{v}_i = \boldsymbol{W}^V\boldsymbol{x}_i$
- Then:
  - $score(\text{input } i, \text{ input } j) = \boldsymbol{q}_i^T\boldsymbol{k}_j$
  - $\alpha_{ij} = softmax(score(input\ i, input\ j))$
  - $\boldsymbol{y}_i = \Sigma_j\, \alpha_{ij}\, \boldsymbol{v}_j$
- Note that for dot product attention, we require that $\boldsymbol{W}^{Q,}\ \boldsymbol{W}^K \in R^{DxD'}$
  - In general, $\boldsymbol{W}^Q$, $\boldsymbol{W}^K$ and $\boldsymbol{W}^V$ can be of any size allowed by the required computations for scores and weights

Self-Attention Layer (3)

- Here, model attends only to previous tokens
  - **Design choice, depends on task**
- Note:
  - **Transformations** to *k*, *q* and *v*
  - **Dot products** with one *q* and each k to compute scores
  - **Softmax** to compute weights
  - **Linear combination** with each *v* to get contextualized representation *y*



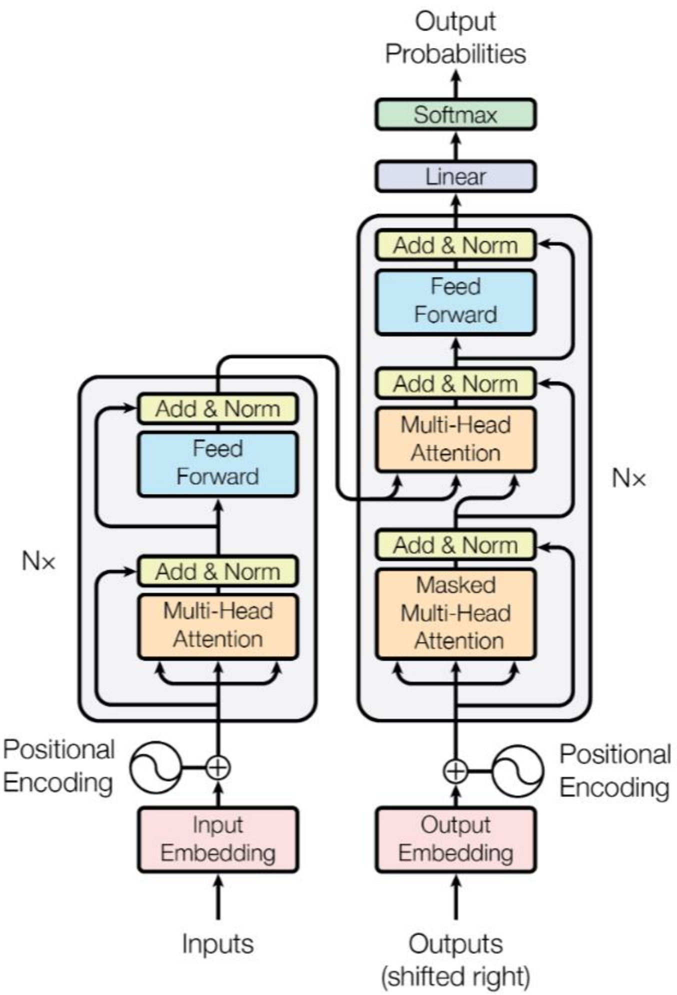
Self-Attention Layer (4)

- The original model actually used scaled dot-product attention
  - *score*(input *i*, input *j*) = $\boldsymbol{q}_i^T\boldsymbol{k}_j / \sqrt{D_k}$ where D_k is size of keys and query vectors
  - Done to avoid numerical stability issues when using softmax
- We can represent the computation for all $\boldsymbol{y}_i$ as matrix products
  - Let $\mathbf{X} \in R^{NxD}$ be the matrix of N input tokens of size D
  - Then: $\boldsymbol{Q} = \boldsymbol{XW}^Q \in R^{NxD'}$, $\boldsymbol{K} = \boldsymbol{XW}^K \in R^{NxD'}$, $\boldsymbol{V} = \boldsymbol{XW}^V \in R^{NxD}$
  - All relevant dot products, i.e. attention scores, are in $\boldsymbol{QK}^T \in R^{NxN}$
  - Apply row-wise softmax for attention weights: $softmax(\boldsymbol{QK}^T / \sqrt{D_k}) \in R^{NxN}$
  - Then multiply by $\boldsymbol{V}$ to get final stacked contextualized representations
- All together: $\text{Self-Attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = softmax(\boldsymbol{QK}^T / \sqrt{D_k})\, \boldsymbol{V}$
- **Cost:** each $\boldsymbol{y}_i$ computed independently, thus highly parallelizable
  - But attention quadratic in input size (every item with every other item)
  - Thus, size of input sequence a fundamental limitation of this architecture
  - Transformers as FNNs, not RNNs: **max. input size fixed**, grows with model

The Transformer Architecture

The Transformer Architecture

- Originally an encoder-decoder architecture
- **Encoder:** multi-head **attention**
  - Plus additional components
- **Decoder:** multi-head **attention**
  - Plus additional components
- **Interaction** between **encoder/decoder**:
  - Multi-head **attention**
- Hence the name: **Attention is All you Need**
- **Multi-head attention:** multiple self-attention layers (discussed soon)
- Additionally, **other components**:
  - Positional encoding
  - Layer normalization
  - Residual connections
  - FNN
- Let's look at them in more detail, but first…

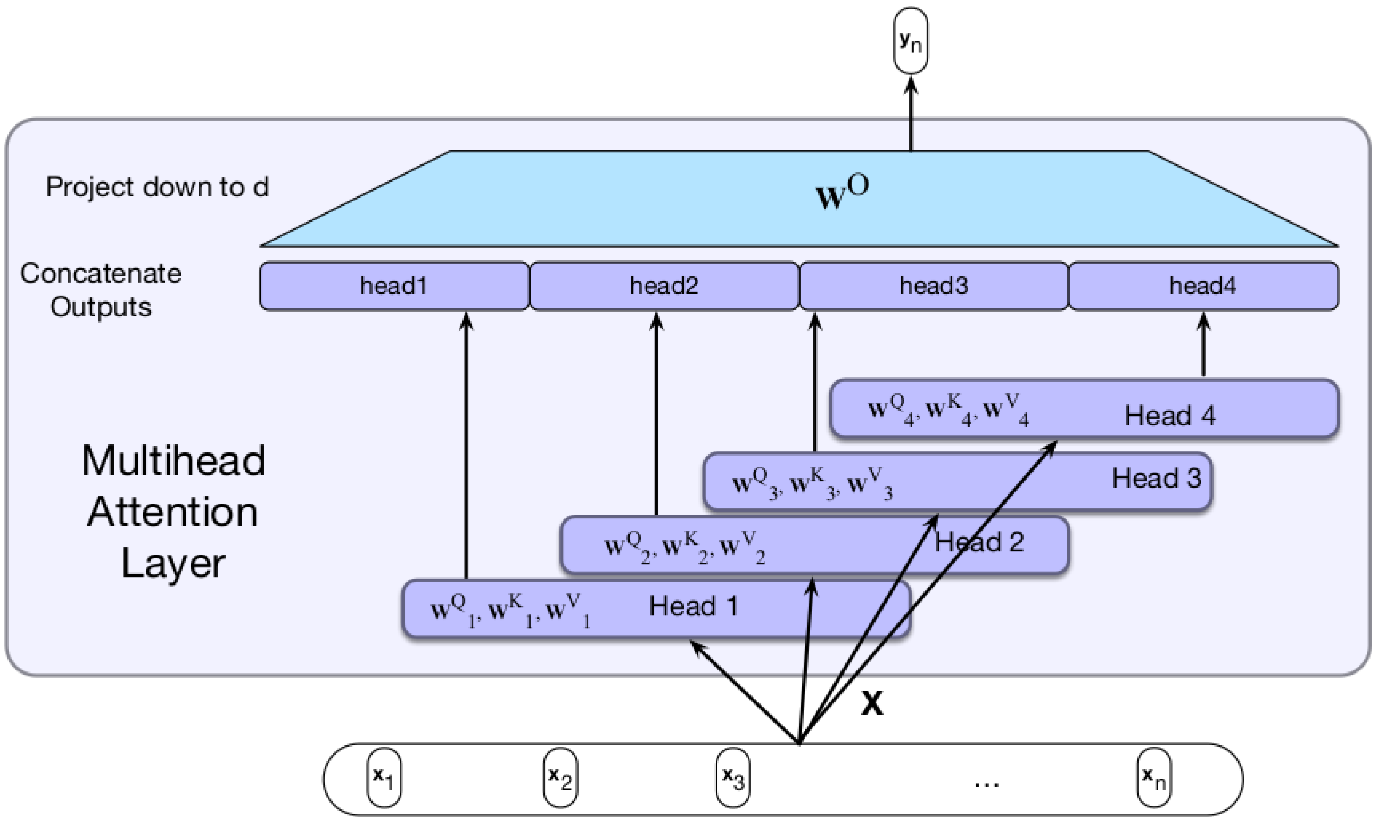


The Animated Transformer

- From Google’s blog, here’s now representations are built/interact
  - 3 encoder layers are stacked for depth (still deep learning)
  - 3 decoder layers are also stacked

Multi-Head Attention (1)

- **Single-head attention:** self-attention layer
- **Multi-head attention:** multiple *independent* self-attention layers
  - Output concatenated, then projected down to input size



Multi-Head Attention (2)

- Specifically, each attention head i has parameters $\boldsymbol{W}_i^Q$, $\boldsymbol{W}_i^K$ and $\boldsymbol{W}_i^V$
  - Since we have a projection layer at the end, dimensions of queries, keys and values can more freely change
  - We denote size of queries and keys by $\boldsymbol{D}_k$ (the same because dot product)
  - We denote size of values by $\boldsymbol{D}_v$
  - Then $\boldsymbol{W}_i^Q \in R^{DxDk}$, $\boldsymbol{W}_i^K \in R^{DxDk}$ and $\boldsymbol{W}_i^V \in R^{DxDv}$ where D is size of input tokens
- As before, stack inputs of size D to form matrix $\boldsymbol{X}$
  - Then, $\boldsymbol{Q}_i = \boldsymbol{X}\boldsymbol{W}_i^Q$, $\boldsymbol{K}_i = \boldsymbol{X}\boldsymbol{W}_i^K$, $\boldsymbol{V}_i = \boldsymbol{X}\boldsymbol{W}_i^V$
- We thus have $\boldsymbol{head}_i = \textit{Self-Attention}(\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i)$
- Outputs of each head is concatenated, projected down to D via $\boldsymbol{W}^O$
  - $\textit{MultiHeadAttention}(\boldsymbol{X}) = (\boldsymbol{head}_1 \, (+) \, \boldsymbol{head}_2 \, (+) \dots (+) \, \boldsymbol{head}_H) \, \boldsymbol{W}^O$, *where (+)* denotes concatenation

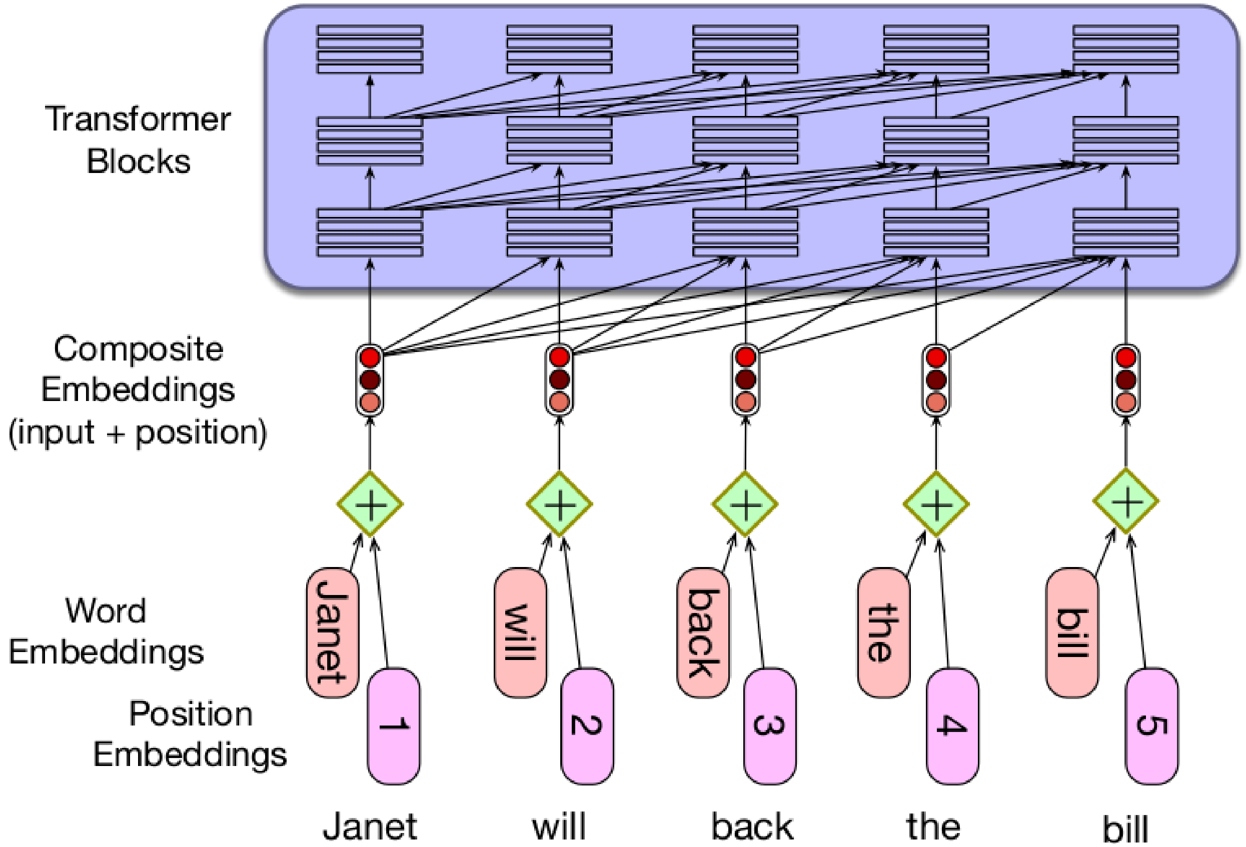
Multi-Head Attention (3)

- Perhaps more important: **why?**
  - What is the **intuition behind this?**
  - How can we think about multi-head attention?
- **Different words in a sentence relate** to each other in **different ways**
  - *"They happily played Mario Kart until the sun came out."*
  - *"Mario Kart"* is the object of verb played
  - *"Mario Kart"* described further by indirect complement *"until the…",* etc.
- A single-head attention must capture all of these dependencies
  - A **multi-head attention splits the workload into independent heads**
  - Think of tasking a set of people to do one job, e.g. memorize a book
- Thus, multi-head attention is more powerful
  - But more difficult to train (more parameters)
- **Additional intuition:** multiple convolution filters in convolution layer
  - Similar principle: many patterns to capture, multiple filters allows model to capture different patterns using different filters

Positional Encoding (1)

- In the transition from RNNs to transformers, we lost something!
  - **We no longer "see" relative positions (order) of each token in the input sequence**
  - Each token attends to each other token, wherever they are
- Think about a self-attention layer
  - It produces contextualized representations for each input token
  - If we **shuffle the input sequence**, the output of the **linear combination** that produces contextualized representations is **still the same**
- But relative position of tokens matters in a sequence!
- **Solution:** modify input embeddings to encode position before applying attention
  - For example, learn position embeddings just as you do word embeddings
  - E.g. embedding for position 1, 2, etc.
  - Then combine embeddings of positions with words before multi-head attention, e.g. by adding them up -> $\boldsymbol{w}_1 + \boldsymbol{w}_{the}$

Positional Encoding (2)



- **Other types of positional encodings**, e.g. non-parameterized ones
  - We'll see some in detail in tutorials

Layer Normalization

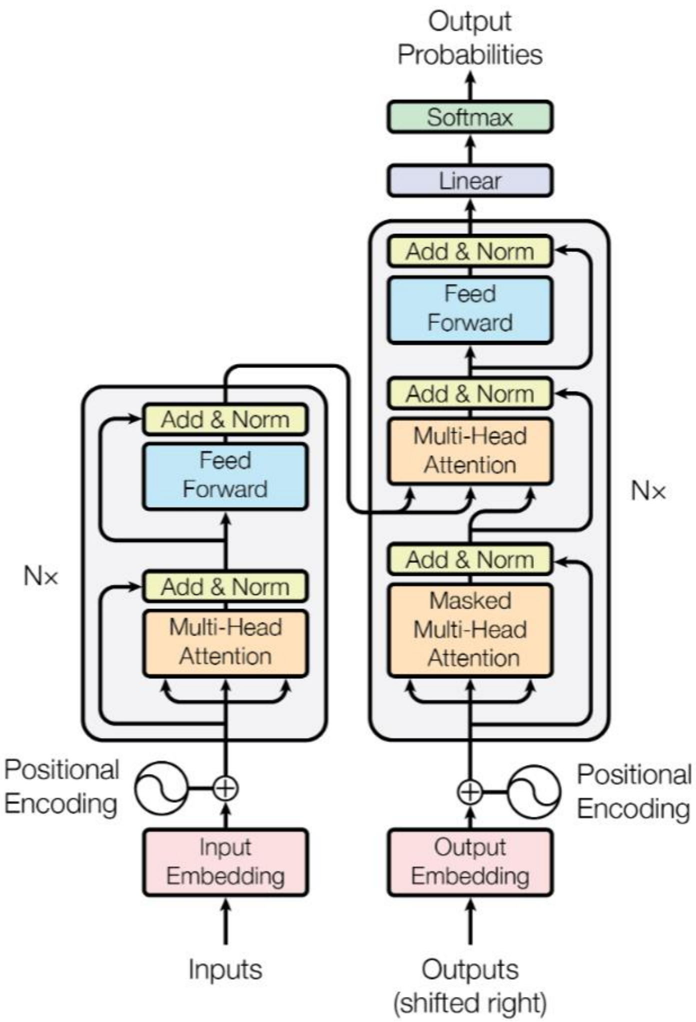
- **Goal:** normalize output of any layer in a network
  - Not exclusive to transformers, commonly used in deep learning models
  - Improves gradient-based training
- It's a form of centering the data
  - Given output vector **x** for some hidden layer, compute its mean μ and standard deviation σ
  - Then center $\mathbf{x'} = (\mathbf{x} - \mu) / \sigma$
- Final value is actually $LayerNorm = \gamma \mathbf{x'} + \beta$
  - Both γ**,** β are learnable parameters
- Layer normalization can play **important role during training**
  - First proposed by Ba et al. (2016)
  - Subsequent studies have focused on its impacts
  - Recent architectures include more layer normalization in other parts
  - E.g. after input embeddings, before and after self-attention
  - More on this when discussing LLMs

Residual Connections

- Recall the **vanishing gradient** problem:
  - Most **operators have impact on** the **gradient that flows backward** during training
  - Generally, **input gradients** are **multiplied by** an operator-specific **Jacobian**
  - This factor **can make output gradient smaller than input gradient**
  - **With depth, gradients can thus vanish**
- Residual connections designed to address this issue ([He et al. 2015](#))
- **Main idea:** given operator, add its input back to its output
  - Addition operator has no impact on gradient, passes it backward unchanged!
  - Also not exclusive to transformers, commonly done in deep models
- In transformers, $\boldsymbol{z} = \mathit{LayerNorm}(\boldsymbol{X} + \mathit{Self\text{-}Attention}(\boldsymbol{X}))$
  - Thus, whatever impact *Self-Attention* has on the gradient, the full gradient is still preserved by the additive term $\boldsymbol{X}$ during backpropagation
  - We discuss this in a bit more detail in tutorials

Projection FNN and Cross-Attention

- Often described as **MLP**
- FNN used to transform output of multihead-attention
  - In reality, two linear projections, usually project up, then back down
  - ReLU (non-linear) activation between
  - Unclear exactly why, more soon
- Important: **cross-attention**
  - Attention across different sequences
  - Note two attention layers in decoder
  - One attends to output of encoder
- Different transformer architectures
  - Encoder-decoder
  - Encoder-only, decoder-only
- More in lecture on LLMs



Dropout

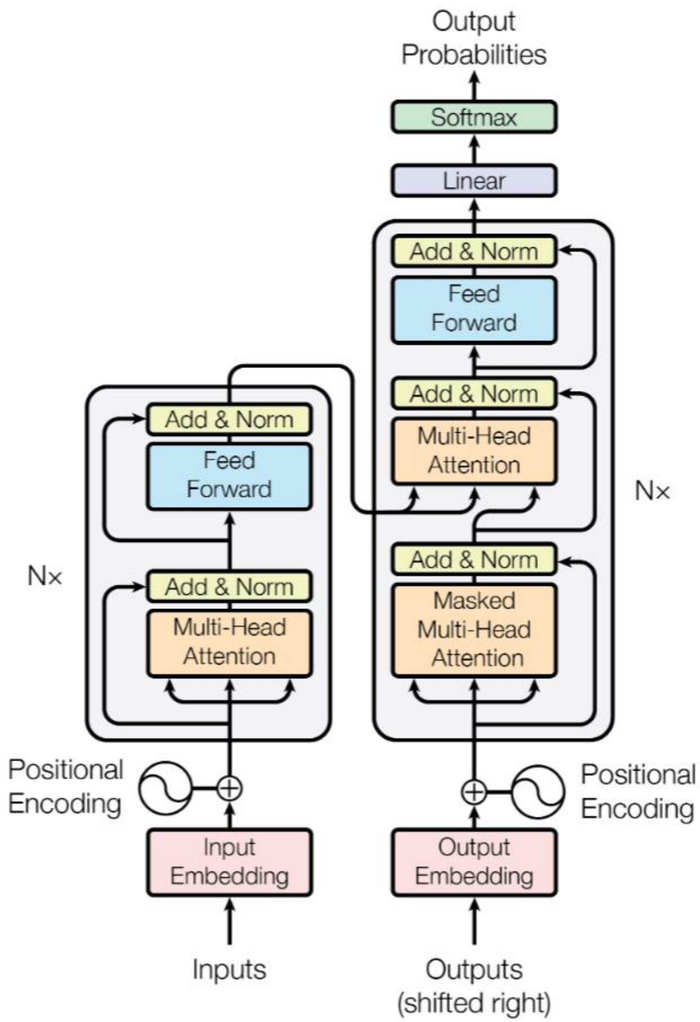
- Proposed by Srivastava et al. (2014)
- **Idea:** randomly drop units in network (along with their connections)
  - It's a form of regularization, thus, only applied during training, not at inference time
  - Intuitively, network is dynamic during training
  - Prevents model from relying too much on static architecture by introducing variability, promoting generalization
- In practice, each units "zeroed" with probability *p* (hyperparameter)
  - E.g. see implementation in PyTorch
- **Original transformer used dropout** in two places
  - Specifically, before applying layer normalization and residual connection
  - That is, $\boldsymbol{z} = LayerNorm(\boldsymbol{X} + Dropout(Self\text{-}Attention(\boldsymbol{X})))$
  - Also after adding positional embeddings, i.e. x $= Dropout(\boldsymbol{x}_1 + \boldsymbol{x})$
- Dropout still very common regularization method in LLMs

Inductive Bias in Transformers

- **Inductive bias:** set of assumptions made by a learning model
- Common story: self-attention is main innovation
  - But many more components in transformers
- **Transformer block:**
  - Two main components: **self-attention and MLP**
  - With corresponding layer normalizations and residual connections
  - Let's see the role of each
- **Self-attention:**
  - Linear operations between different input tokens
  - I.e. there is **"inter-token"** communication
  - Linearities useful for parallelization, but not expressive for learning
- **MLP:**
  - Transformations applied at each token independently
  - I.e. **"Intra-token"** phase
  - Non-linearities introduced here for expressivity

Ablation

- There is **no fundamental theory behind the design of deep learning models**
  - No way to accurately predict how a model will behave given its architecture
  - No equation that tells us exactly which components a model should have to perform a given task
- **Thus, ablation studies are essential!**
  - **Ablation:** remove one component of the model at a time to test its impact
  - E.g. remove normalization layer, run model again, how does performance change?
  - Or remove/change positional encodings. What impact does this have in performance?
- With ablation, we may gain some understanding of why a model works



Summary

- **Self-attention:** main innovation in transformer architecture
  - Each input token contextualized based on entire input sequence
- Architecture has **many components**
  - Self-attention
  - Positional encodings
  - Layer normalization
  - FNN
- Architectures are **normally deep** (stacked transformer blocks)
- Different flavors
  - Encoder-decoder, decoder-only, etc.
- **Large language models (LLMs)** based on transformers, with variations
  - More layer normalizations
  - Different positional embeddings
  - Etc.

References

- Speech and Language Processing, Jurafsky et al., 2024
  - Chapter 10
- References linked in corresponding slides