Lecture 8. Transformers

Maybe attention is all you need

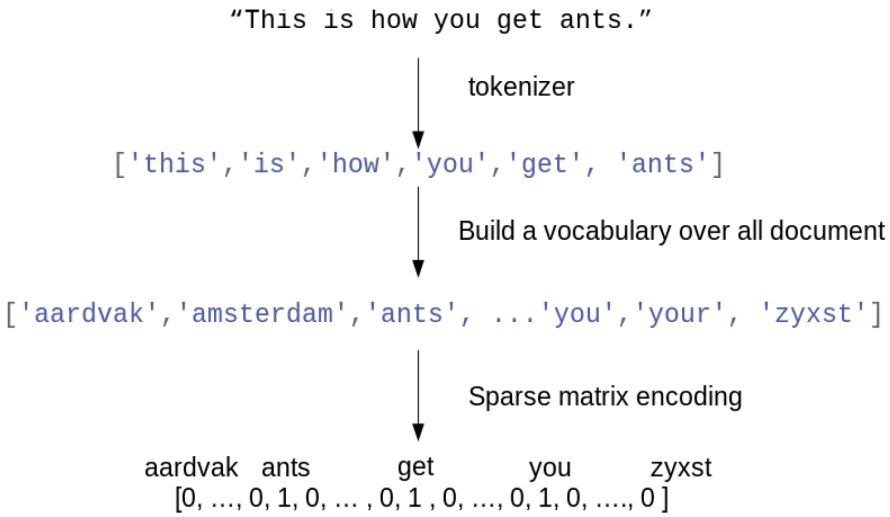
Joaquin Vanschoren

Overview

- Basics: word embeddings
    - Word2Vec, FastText, GloVe
- Sequence-to-sequence and autoregressive models
- Self-attention and transformer models
- Vision Transformers

Bag of word representation

- First, build a *vocabulary* of all occuring words. Maps every word to an index.
- Represent each document as an N dimensional vector (top-N most frequent words)
    - One-hot (sparse) encoding: 1 if the word occurs in the document
- Destroys the order of the words in the text (hence, a 'bag' of words)



Text preprocessing pipelines

- Tokenization: how to you split text into words / tokens?
- Stemming: naive reduction to word stems. E.g. 'the meeting' to 'the meet'
- Lemmatization: NLP-based reduction, e.g. distinguishes between nouns and verbs
- Discard stop words ('the', 'an',...)
- Only use N (e.g. 10000) most frequent words, or a hash function
- n-grams: Use combinations of n adjacent words next to individual words
    - e.g. 2-grams: "awesome movie", "movie with", "with creative", ...
- Character n-grams: combinations of n adjacent letters: 'awe', 'wes', 'eso',...
- Subword tokenizers: graceful splits "unbelievability" -> un, believ, abil, ity
- Useful libraries: nltk, spaCy, gensim, HuggingFace tokenizers,...

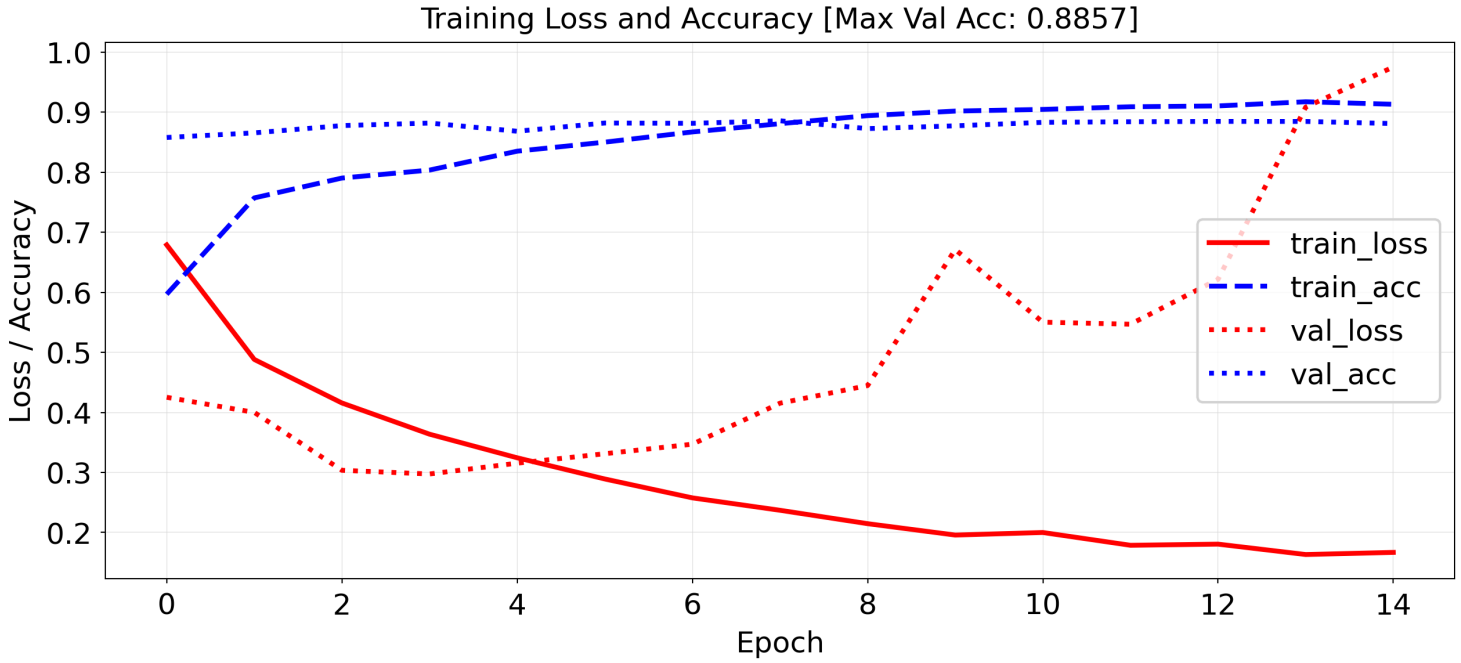
Neural networks on bag of words

- We can build neural networks on bag-of-word vectors
    - Do a one-hot-encoding with 10000 most frequent words
    - Simple model with 2 dense layers, ReLU activation, dropout

```
self.model = nn.Sequential(
    nn.Linear(10000, 16),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(16, 16),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(16, 1)
)
```

Evaluation

- IMDB dataset of movie reviews (label is 'positive' or 'negative')
- Take a validation set of 10,000 samples from the training set
- Works prety well (88% Acc), but overfits easily




```
`Trainer.fit` stopped: `max_epochs=15` reached.
```

Predictions

Let's look at a few predictions. Why is the last one so negative?

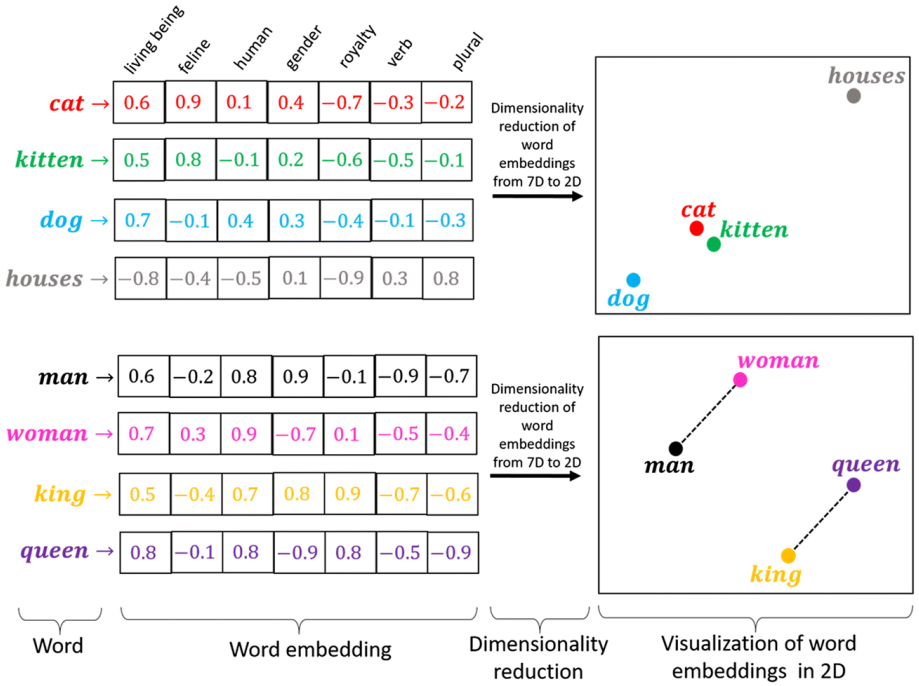
```
Review 0:
 [START] please give this one a miss br br [UNK] [UNK] and the rest of t
he cast rendered terrible performances the show is flat flat flat br br
i don't know how michael madison could have allowed this one on his plat
e he almost seemed to know this wasn't going to work out and his perform
ance was quite [UNK] so all you madison fans give this a miss
Predicted positiveness: 0.15110373

Review 16:
 [START] from 1996 first i watched this movie i feel never reach the end
of my satisfaction i feel that i want to watch more and more until now m
y god i don't believe it was ten years ago and i can believe that i almo
st remember every word of the dialogues i love this movie and i love thi
s novel absolutely perfection i love willem [UNK] he has a strange voice
to spell the words black night and i always say it for many times never
being bored i love the music of it's so much made me come into another w
orld deep in my heart anyone can feel what i feel and anyone could make
the movie like this i don't believe so thanks thanks
Predicted positiveness: 0.99687344

Review X:
 [START] the restaurant is not too terrible
Predicted positiveness: 0.8728
```

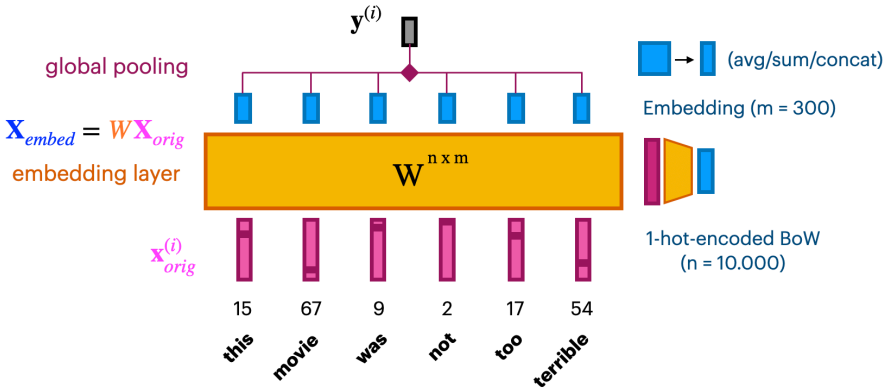
Word Embeddings

- A word embedding is a numeric vector representation of a word
  - Can be manual or *learned* from an existing representation (e.g. one-hot)



Learning embeddings from scratch

- Input layer uses fixed length documents (with 0-padding).
- Add an *embedding layer* to learn the embedding
    - Create n-dimensional one-hot encoding.
    - To learn an m-dimensional embedding, use m hidden nodes. Weight matrix W^{nxm}
    - Linear activation function: $\mathbf{X}_{embed} = W\mathbf{X}_{orig}$.
- Combine all word embeddings into a document embedding (e.g. global pooling).
- Add layers to map word embeddings to the output. Learn embedding weights from data.

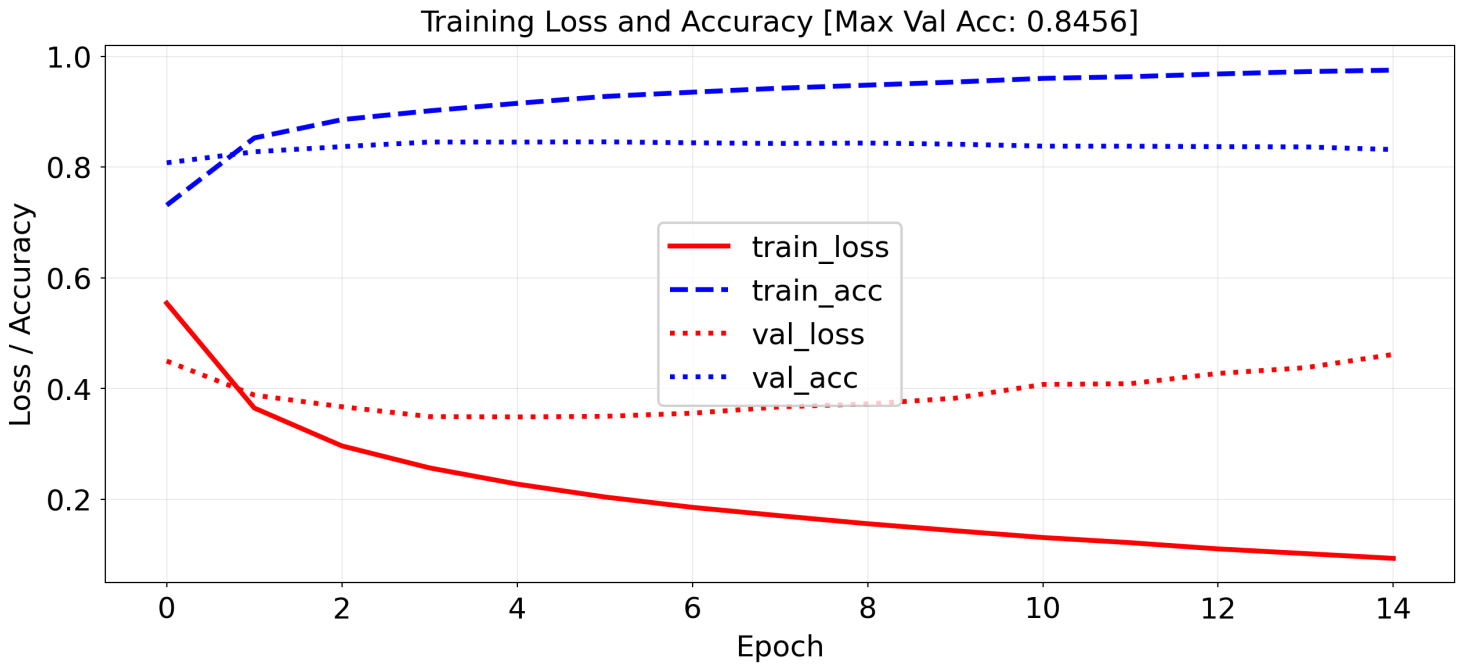


Let's try this:

```
max_length = 100 # pad documents to a maximum number of words
vocab_size = 10000 # vocabulary size
embedding_length = 20 # embedding length (more would be better)

self.model = nn.Sequential(
    nn.Embedding(vocab_size, embedding_length),
    nn.AdaptiveAvgPool1d(1),  # global average pooling over sequence
    nn.Linear(embedding_length, 1),
)
```

- Training on the IMDB dataset: slightly worse than using bag-of-words?
    - Embedding of dim 20 is very small, should be closer to 100 (or 300)
    - We don't have enough data to learn a really good embedding from scratch



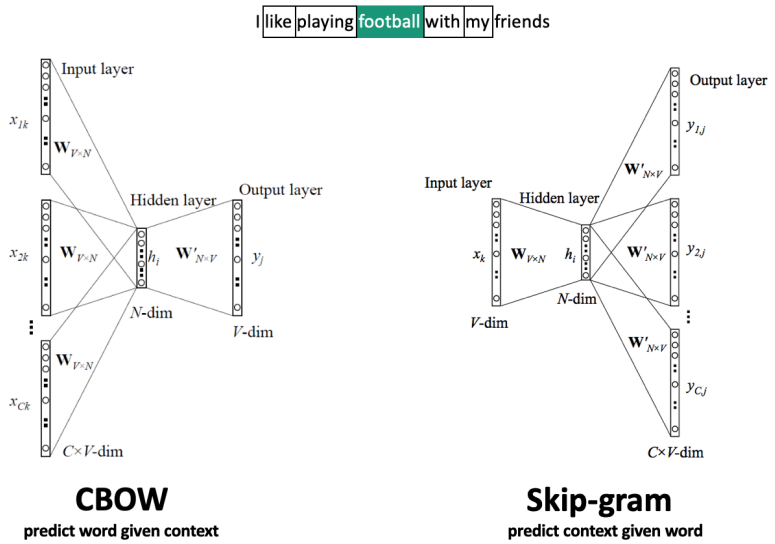

```
`Trainer.fit` stopped: `max_epochs=15` reached.
```

Pre-trained embeddings

- With more data we can build better embeddings, but we also need more labels
- Solution: transfer learning! Learn embedding on auxiliary task that doesn't require labels
    - E.g. given a word, predict the surrounding words.
    - Also called self-supervised learning. Supervision is provided by data itself
- Freeze embedding weights to produce simple word embeddings, or finetune to a new tasks
- Most common approaches:
    - Word2Vec: Learn neural embedding for a word based on surrounding words
    - FastText: learns embedding for character n-grams
        - Can also produce embeddings for new, unseen words
    - GloVe (Global Vector): Count co-occurrences of words in a matrix
        - Use a low-rank approximation to get a latent vector representation

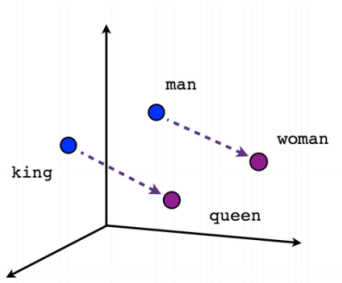
Word2Vec

- Move a window over text to get C context words (V-dim one-hot encoded)
- Add embedding layer with N linear nodes, global average pooling, and softmax layer(s)
- CBOW: predict word given context, use weights of last layer W'_{NxV} as embedding
- Skip-Gram: predict context given word, use weights of first layer W^T_{VxN} as embedding
    - Scales to larger text corpora, learns relationships between words better

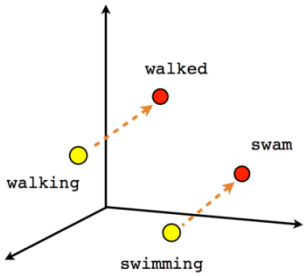


Word2Vec properties

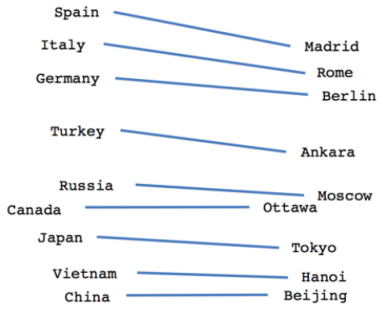
- Word2Vec happens to learn interesting relationships between words
    - Simple vector arithmetic can map words to plurals, conjugations, gender analogies,...
    - e.g. Gender relationships: $vec_{king} - vec_{man} + vec_{woman} \sim vec_{queen}$
    - PCA applied to embeddings shows Country - Capital relationship
- Careful: embeddings can capture gender and other biases present in the data.
    - Important unsolved problem!



Male-Female



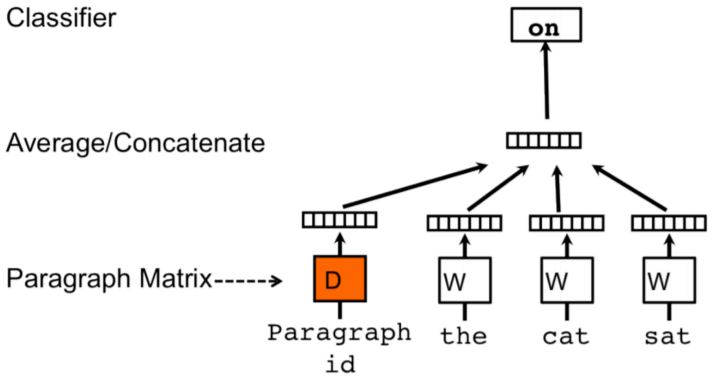
Verb tense



Country-Capital

Doc2Vec

- Alternative way to combine word embeddings (instead of global pooling)
- Adds a paragraph (or document) embedding: learns how paragraphs (or docs) relate to each other
    - Captures document-level semantics: context and meaning of entire document
- Can be used to determine semantic similarity between documents.



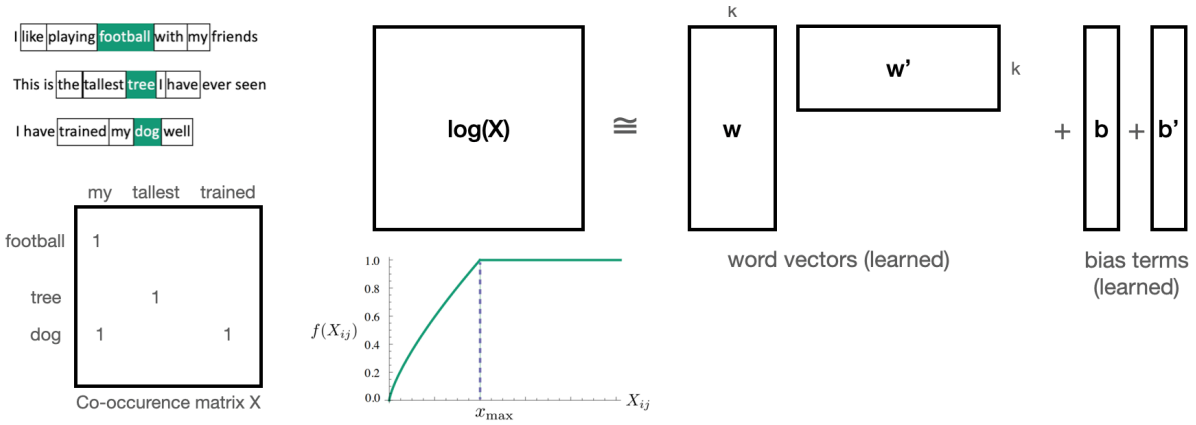
FastText

- Limitations of Word2Vec:
    - Cannot represent new (out-of-vocabulary) words
    - Similar words are learned independently: less efficient (no parameter sharing)
        - E.g. 'meet' and 'meeting'
- FastText: same model, but uses *character n-grams*
    - Words are represented by all character n-grams of length 3 to 6
        - "football" 3-grams: <fo, foo, oot, otb, tba, bal, all, ll>
    - Because there are so many n-grams, they are hashed (dimensionality = bin size)
    - Representation of word "football" is sum of its n-gram embeddings
- Negative sampling: also trains on random negative examples (out-of-context words)
    - Weights are updated so that they are *less* likely to be predicted

Global Vector model (GloVe)

- Builds a co-occurence matrix $\mathbf{X}$: counts how often 2 words occur in the same context
- Learns a k-dimensional embedding W through matrix factorization with rank k
    - Actually learns 2 embeddings W and W' (differ in random initialization)
- Minimizes loss $\mathcal{L}$, where b_i and b'_i are bias terms and f is a weighting function

$$\mathcal{L} = \sum_{i,j=1}^{V} f(\mathbf{X}_{ij})(\mathbf{w_i}\mathbf{w'_j} + b_i + b'_j - log(\mathbf{X}_{ij}))^2$$



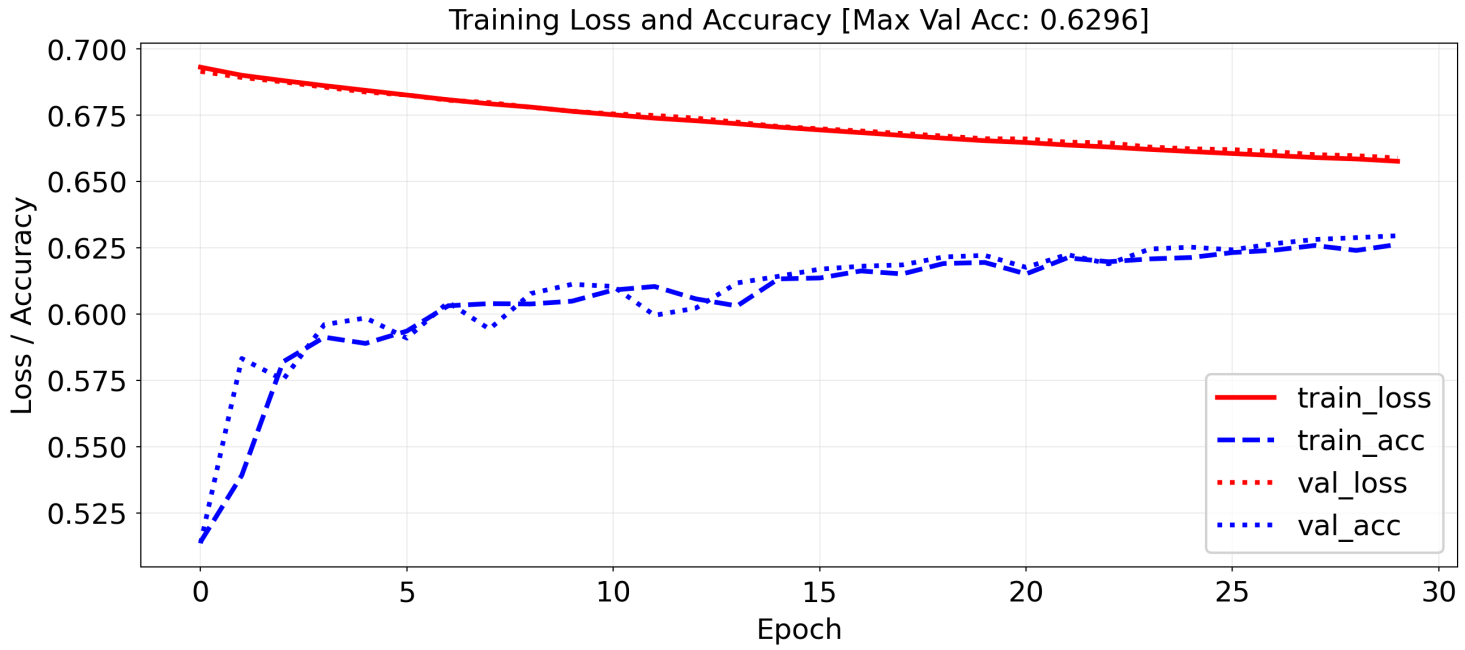
Let's try this

- Download the GloVe embeddings trained on Wikipedia
- We can now get embeddings for 400,000 English words
- E.g. 'queen' (in 300-dim):

```
array([-0.222,  0.065, -0.086,  0.513,  0.325, -0.129,  0.083,  0.092,
       -0.309, -0.941, -0.089, -0.108,  0.211,  0.701,  0.268, -0.04 ,
        0.174, -0.308, -0.052, -0.175, -0.841,  0.192, -0.138,  0.385,
        0.272, -0.174, -0.466, -0.025,  0.097,  0.301,  0.18 , -0.069,
       -0.205,  0.357, -0.283,  0.281, -0.012,  0.107, -0.244, -0.179,
       -0.132, -0.17 , -0.594,  0.957,  0.204, -0.043,  0.607, -0.069,
        0.523, -0.548, -0.545,  0.435,  0.445, -0.026, -0.115, -0.186,
        0.361, -0.251,  0.721, -0.058,  0.344,  0.472,  0.205,  0.7  ,
        0.487, -0.903, -0.104,  0.127, -0.411,  0.404,  0.096,  0.332,
        0.07 , -0.197,  0.159,  0.097,  0.558, -0.549,  0.092,  0.234,
        0.395, -0.11 ,  0.106, -0.258,  0.556, -0.54 ,  0.102, -0.033,
        0.109, -0.446,  0.228, -0.424,  0.08 , -0.437, -0.203, -0.422,
        0.525, -0.133, -0.048, -0.555,  0.273, -0.361,  0.392,  0.064,
        0.195,  0.427,  0.06 ,  0.11 ,  0.05 ,  0.289,  0.288,  0.685,
        0.41 , -0.253,  0.059, -0.176,  0.086,  0.17 , -0.083, -0.57 ,
       -0.719,  0.703, -0.379, -0.477, -0.097,  0.14 , -0.019, -0.148,
        0.487,  0.448,  0.574,  0.54 , -0.139, -0.141,  0.194, -0.009,
       -0.233,  0.051, -0.429,  0.584,  0.209,  0.751, -0.661, -0.151,
        0.202, -0.54 ,  0.413,  0.361, -0.382, -0.39 ,  0.154, -0.211,
       -0.562,  0.519,  0.224,  0.554,  0.058, -0.434,  0.087, -0.551,
        0.582,  0.828,  0.506,  0.146, -0.36 ,  0.825, -0.253,  0.666,
        0.329,  0.382, -0.51 ,  0.908, -0.067, -0.621, -0.229,  0.309,
```

- Same simple model, but with frozen GloVe embeddings: much worse!
- Linear layer is too simple. We need something more complex -> transformers :)

```
embedding_tensor = torch.tensor(embedding_matrix, dtype=torch.float32)
self.model = nn.Sequential(
    nn.Embedding.from_pretrained(embedding_tensor, freeze=True),
    nn.AdaptiveAvgPool1d(1),
    nn.Linear(embedding_tensor.shape[1], 1))
```



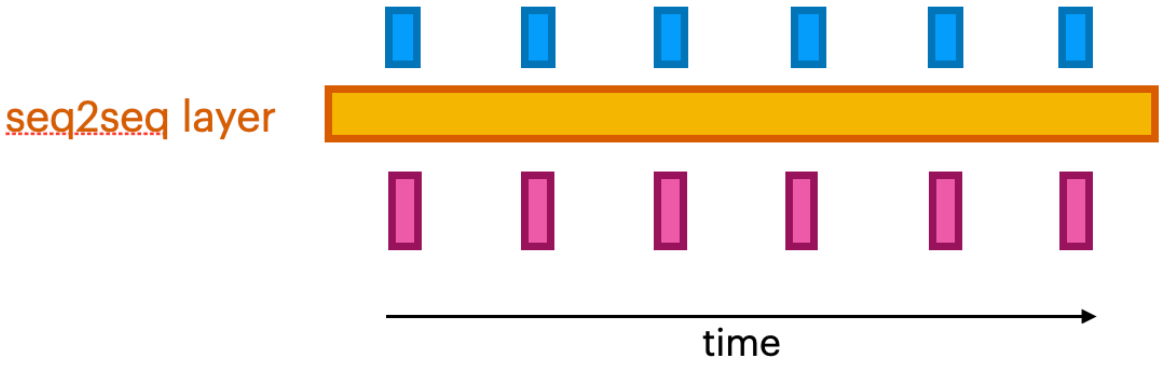

```
`Trainer.fit` stopped: `max_epochs=30` reached.
```

Sequence-to-sequence (seq2seq) models

- Global average pooling or flattening destroys the word order
- We need to model sequences explictly, e.g.:
    - 1D convolutional models: run a 1D filter over the input data
        - Fast, but can only look at small part of the sentence
    - Recurrent neural networks (RNNs)
        - Can look back at the entire previous sequence
        - Much slower to train, have limited memory in practice
    - Attention-based networks (Transformers)
        - Best of both worlds: fast and very long memory

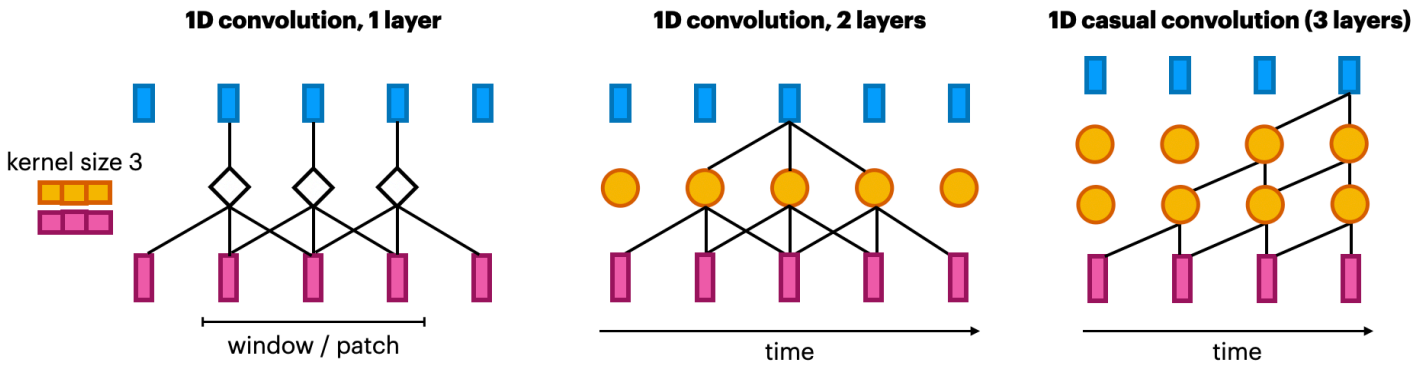
seq2seq models

- Produce a series of output given a series of inputs over time
- Can handle sequences of different lengths
    - Label-to-sequence, Sequence-to-label, seq2seq,...
    - Autoregressive models (e.g. predict the next character, unsupervised)



1D convolutional networks

- Similar to 2D convnets, but moves only in 1 direction (time)
    - Extract local 1D patch, apply filter (kernel) to every patch
    - Pattern learned can later be recognized elsewhere (translation invariance)
- Limited memory: only sees a small part of the sequence (receptive field)
    - You can use multiple layers, dilations,... but becomes expensive
- Looks at 'future' parts of the series, but can be made to look only at the past
    - Known as 'causal' models (not related to causality)

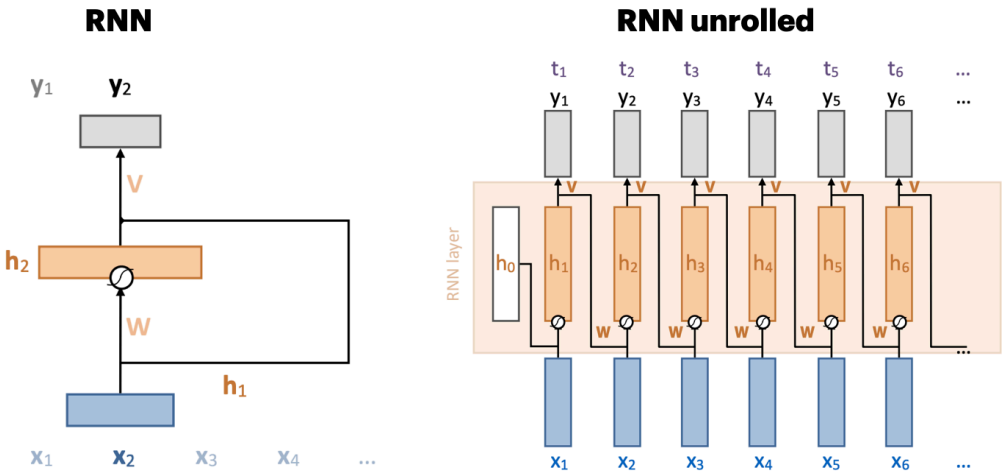


- Same embedding, but add 2 `Conv1D` layers and `MaxPooling1D`. Better!

```
model = nn.Sequential(
    nn.Embedding(num_embeddings=10000, embedding_dim=embedding_dim),
    nn.Conv1d(in_channels=embedding_dim, out_channels=32, kernel_size=7),
    nn.ReLU(),
    nn.MaxPool1d(kernel_size=5),
    nn.Conv1d(in_channels=32, out_channels=32, kernel_size=7),
    nn.ReLU(),
    nn.AdaptiveAvgPool1d(1),  # GAP
    nn.Flatten(),             # (batch, 32, 1) → (batch, 32)
    nn.Linear(32, 1)
)
```

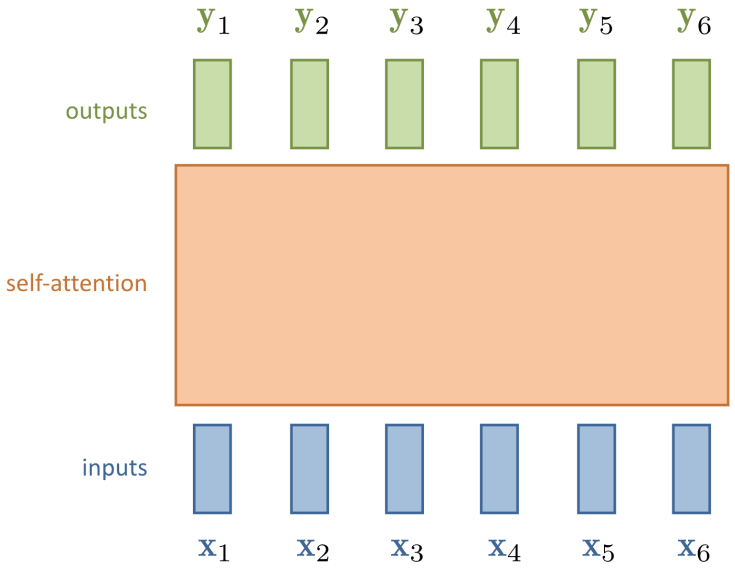
Recurrent neural networks (RNNs)

- Recurrent connection: concats output to next input $h_t = \sigma\left(W \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} + b\right)$
- Unbounded memory, but training requires *backpropagation through time*
    - Requires storing previous network states (slow + lots of memory)
    - Vanishing gradients strongly limit practical memory
- Improved with *gating*: learn what to input, forget, output (LSTMs, GRUs,...)



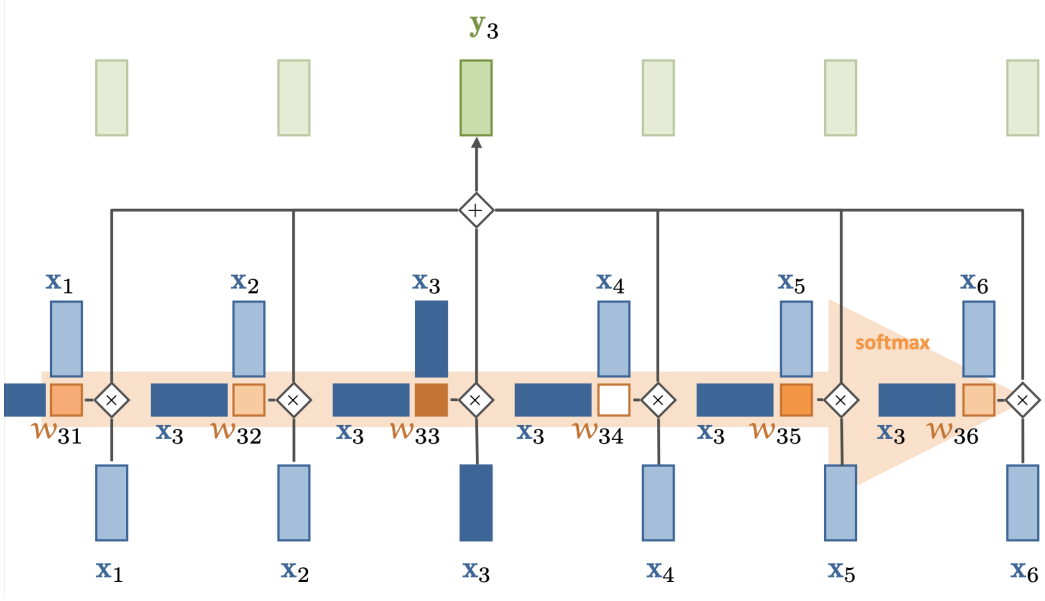
Simple self-attention

- Maps a *set* of inputs to a set of outputs (*without* learned weigths)

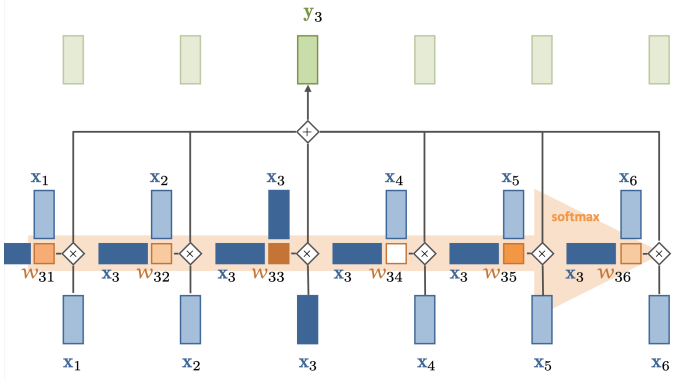


Simple self-attention

- Compute dot product of input vector x_i with every x_j (including itself): w_{ij}
- Compute softmax over all these weights (positive, sum to 1)
- Multiply by each input vector, and sum everything up
- Can be easily vectorized: $Y^T = WX^T$, $W = \text{softmax}(X^T X)$



- For each output, we mix information from all inputs according to how 'similar' they are
    - The set of weights w_i for a given token is called the attention vector
    - It says how much 'attention' each token gives to other tokens
- Doesn't learn (no parameters), the embedding of X defines self-attention
    - We'll learn how to *transform the embeddings* later
    - That way we can learn different relationships (not just similarity)
- Has no problem looking *very* far back in the sequence
- Operates on *sets* (permutation invariant): allows img-to-set, set-to-set,... tasks
    - If the token order matters, we'll have to encode it in the token embedding

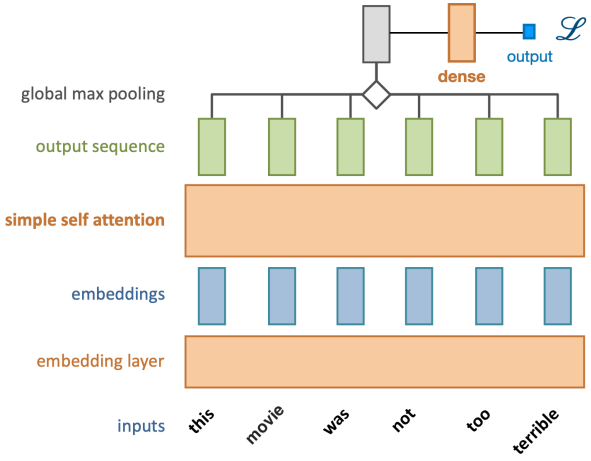


Scaled dot products

- Self-attention is powerful because it's mostly a linear operation
- $Y^T = WX^T$ is *linear*, there are no vanishing gradients
    - The softmax function only applies to $W = \mathrm{softmax}(X^T X)$, not to Y^T
    - Needed to make the attention values sum up nicely to 1 without exploding
- The dot products do get larger as the embedding dimension k gets larger (by a factor $\sqrt{k}$)
    - We therefore normalize the dot product by the input dimension k: $w'_{ij} = \frac{{x_i}^T x_j}{\sqrt{k}}$
    - This also makes training more stable: large softmas values lead to 'sharp' outputs, making some gradients very large and others very small

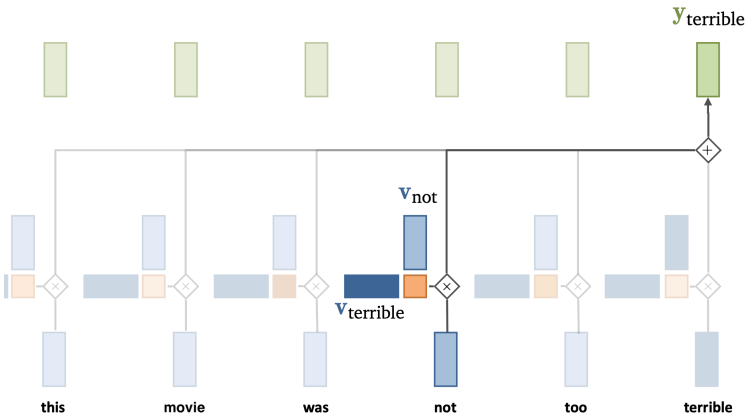
Simple self-attention layer

- Let's add a simple self-attention layer to our movie sentiment model
- Without self-attention, every word would contribute independently (bag of words)
    - The word *terrible* will likely result in a negative prediction
- Now, we can freeze the embedding, take output Y, obtain a loss, and do backpropagation so that the self-attention layer can *learn* that 'not' should invert the meaning of 'terrible'



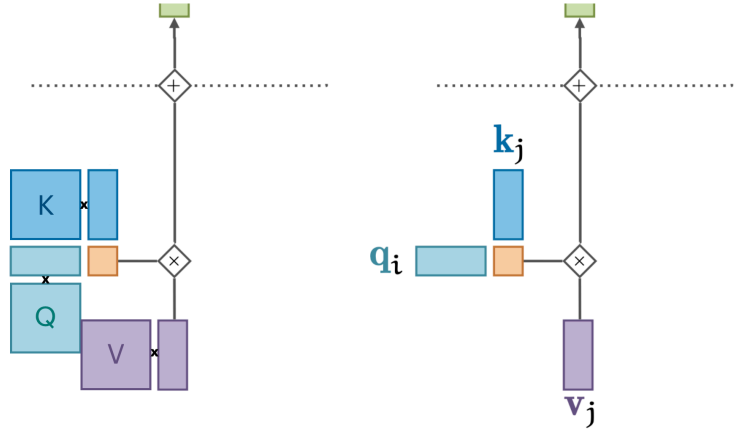
Simple self-attention layer

- Through training, we want the self-attention to learn how certain tokens (e.g. 'not') can affect other tokens / words.
    - E.g. we need to learn to change the representations of v_{not} and $v_{terrible}$ so that they produce a 'correct' (low loss) output
- For that, we do need to add some trainable parameters.



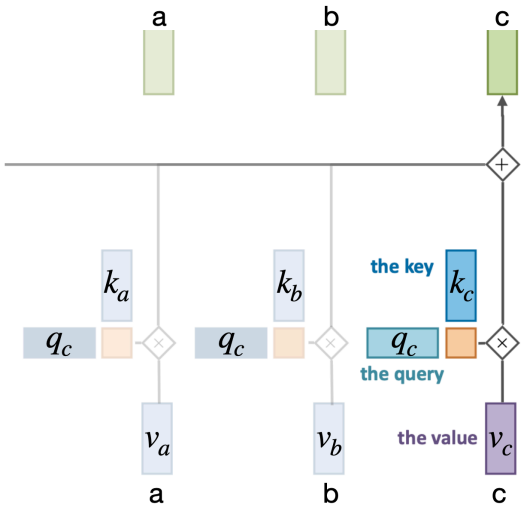
Standard self-attention

- We add 3 weight matrices (K, Q, V) and biases to change each vector:
  - $k_i = Kx_i + b_k$
  - $q_i = Qx_i + b_q$
  - $v_i = Vx_i + b_v$
- The same K, Q, V are used for all tokens depending on whether they are the input token (v), the token we are currently looking at (q), or the token we're comparing with (k)



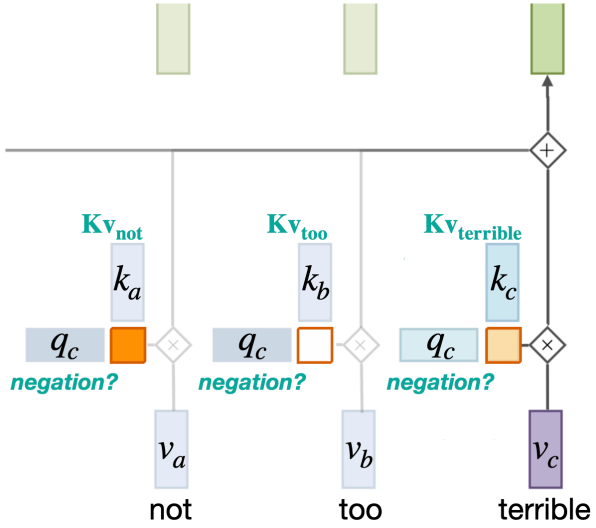
Sidenote on terminology

- View the set of tokens as a dictionary `s = {a: v_a, b: v_b, c: v_c}`
- In a dictionary, the third output (for key c) would simple be `s[c] = v_c`
- In a *soft* dictionary, it's a weighted sum: $s[c] = w_a * v_a + w_b * v_b + w_c * v_c$
- If w_i are dot products: $s[c] = (k_a \cdot q_c) * v_a + (k_b \cdot q_c) * v_b + (k_b \cdot q_c) * v_c$
- We *blend* the influence of every token based on their learned relations with other tokens



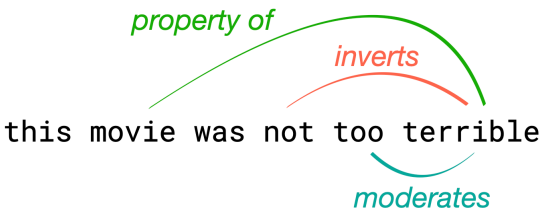
Intuition

- We *blend* the influence of every token based on their learned 'relations' with other tokens
- Say that we need to learn how 'negation' works
    - The 'query' vector could be trained (via Q) to say something like 'are there any negation words?'
    - A token (e.g. 'not'), transformed by K, could then respond very positively if it is



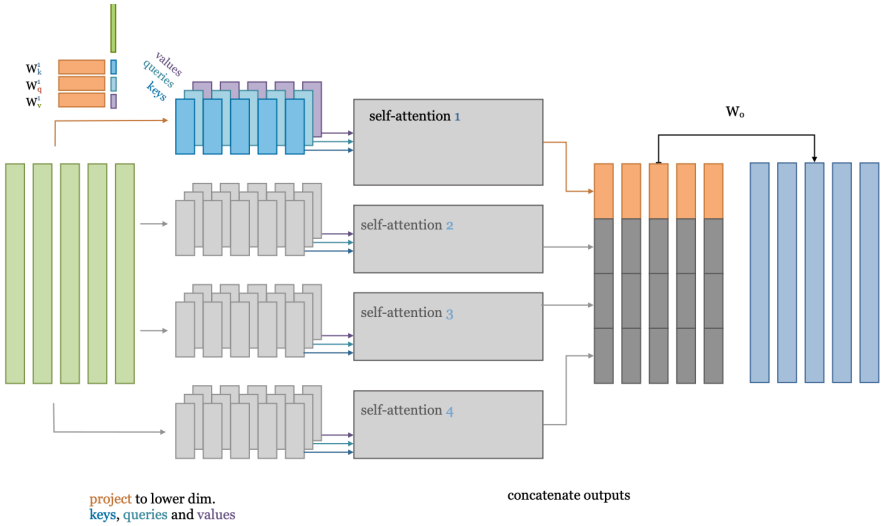
Single-head self-attention

- There are different relations to model within a sentence.
- The same input token, e.g. $v_{terrible}$ can relate completely differently to other kinds of tokens
    - But we only have one set of K, V, and Q matrices
- To better capture multiple relationships, we need multiple self-attention operations (expensive)



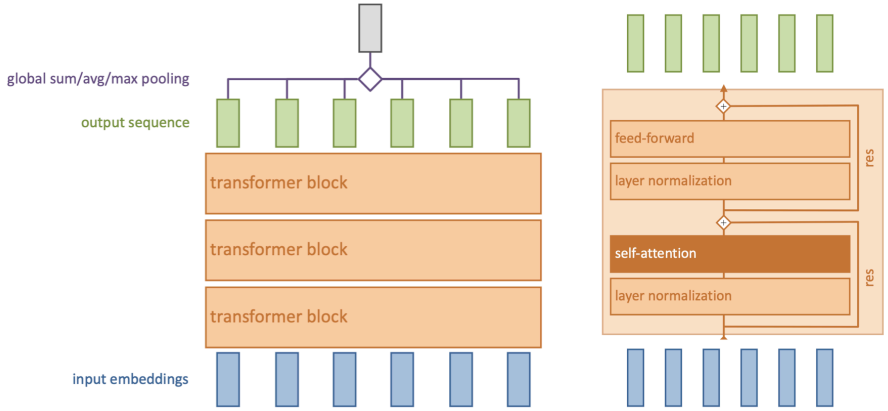
Multi-head self-attention

- What if we project the input embeddings to a lower-dimensional embedding k?
- Then we could learn multiple self-attention operations in parallel
- Effectively, we split the self-attention in multiple heads
    - Each applies a separate low-dimensional self attention (with $K^{kxk}, Q^{kxk}, V^{kxk}$)
- After running them (in parallel), we concatenate their outputs.



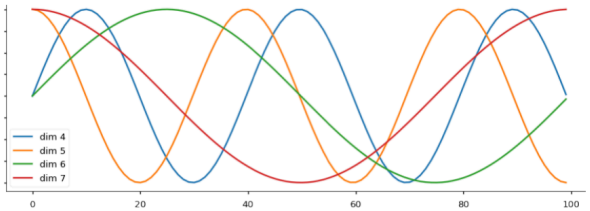
Transformer model

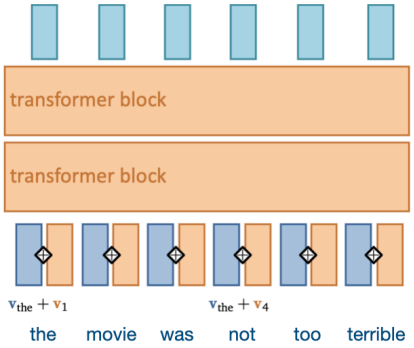
- Repeat self-attention multiple times in controlled fashion
- Works for sequences, images, graphs,... (learn how sets of objects interact)
- Models consist of multiple transformer blocks, usually:
    - Layer normalization (every input is normalized independently)
    - Self-attention layer (learn interactions)
    - Residual connections (preserve gradients in deep networks)
    - Feed-forward layer (learn mappings)



Positional encoding

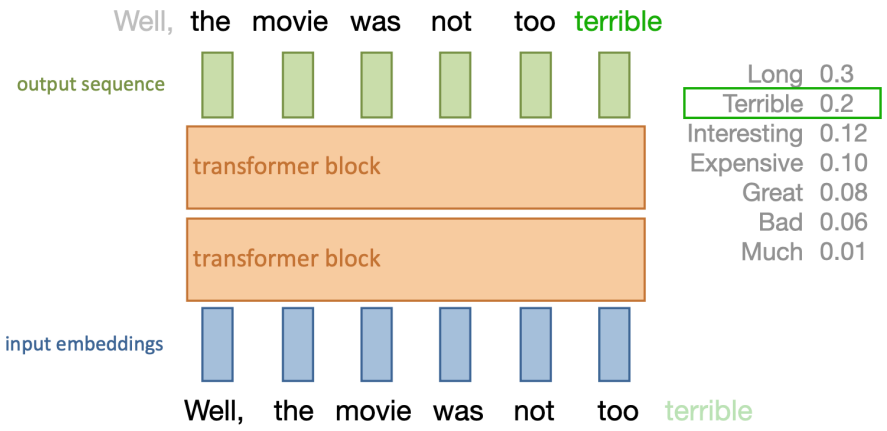
- We need some way to tell the self-attention layer about position in the sequence
- Represent position by vectors, using some easy-to-learn predictable pattern
    - Add these encodings to vector embeddings
    - Gives information on how far one input is from the others
- Other techniques exist (e.g. relative positioning)





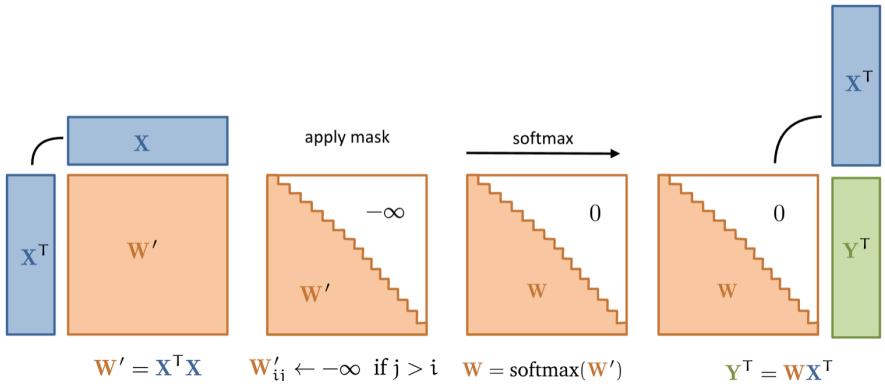
Autoregressive models

- Models that predict future values based on past values of the same stream
- Output token is mapped to list of probabilities, sampled with softmax (with temperature)
- Problem: self-attention can simply look ahead in the stream
    - We need to make the transformer blocks *causal*



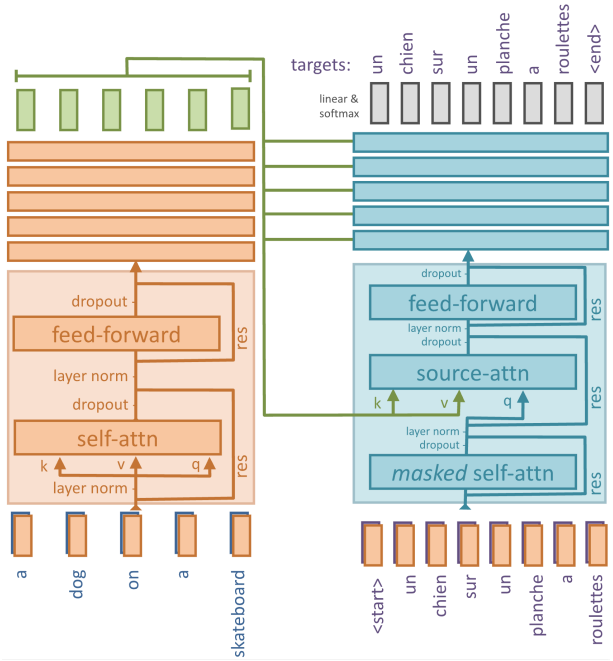
Masked self-attention

- Simple solution: simply mask out any attention weights from current to future tokens
- Replace with -infinity, so that after softmax they will be 0



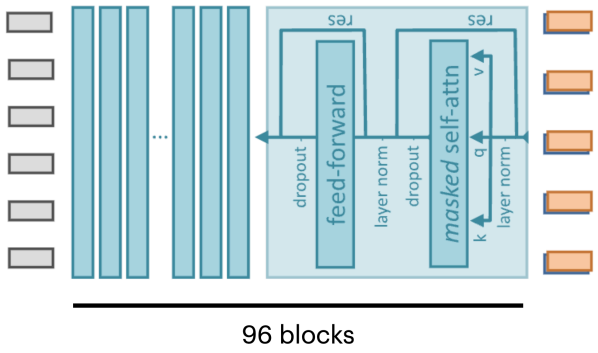
Famous transformers

- "Attention is all you need": first paper to use attention without CNNs or RNNs
- Encoder-Decoder architecture for translation: (k, q) to source attention layer
- We'll reproduce this (partly) in the Lab 6 tutorial :)



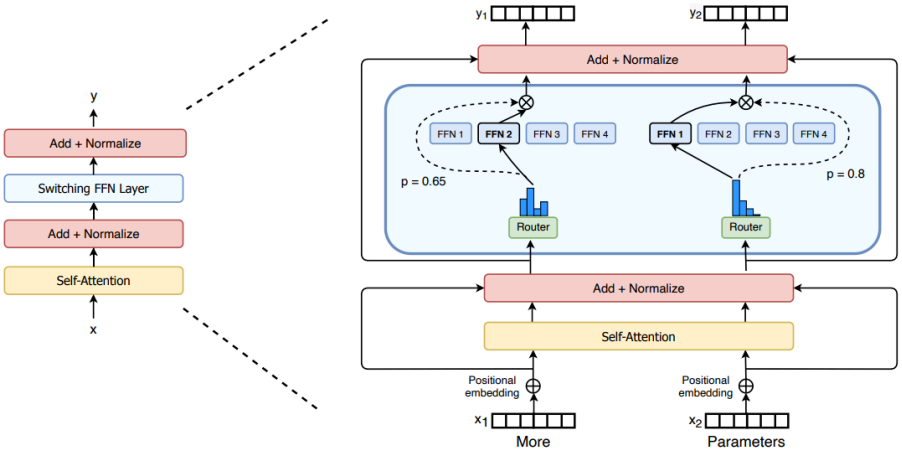
GPT 3

- Decoder-only, single stack of 96 transformer blocks (and 96 heads)
- Sequence size 2048, input dimensionality 12,288, 175B parameters
- Trained on entire common crawl dataset (1 epoch)
    - Additional training on high-quality data (Wikipedia,...)
- Excellent animation by 1b3b
- GPT from scratch by A. Karpathy



GPT 4

- Likely a 'mixtures of experts' model
    - Router (small MLP) selects which subnetworks (e.g. 2) to use given input
    - Predictions get ensembled
- Allows scaling up parameter count without proportionate (inference) cost
- Also better data, more human-in-the-loop training (RLHF),...



Vision transformers

- Same principle: split up into patches, embed into tokens, add position encoding
- For classification: add an extra (random) input token -> [CLS] output token

Demonstration

We'll experiment with the CIFAR-10 datasets

- ViTs are quite expensive on large images.
- This ViT takes about an hour to train (we'll run it from a checkpoint)

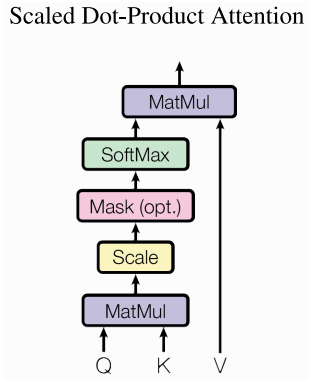
Patchify

- Split $N \times N$ image into $(N/M)^2$ patches of size $M \times M$.

```
B, C, H, W = x.shape  # Batch size, Channels, Height, Width
x = x.reshape(B, C, H//patch_size, patch_size, W//patch_size, patch_size)
```

Self-attention

First, we need to implement a (scaled) dot-product



Self-attention

First, we need to implement a (scaled) dot-product

```
def scaled_dot_product(q, k, v):
    attn_logits = torch.matmul(q, k.transpose(-2, -1)) # dot prod
    attn_logits = attn_logits / math.sqrt(q.size()[-1])# scaling
    attention = F.softmax(attn_logits, dim=-1)         # softmax
    values = torch.matmul(attention, v)                # dot prod
    return values, attention
```

Multi-head attention (simplified)

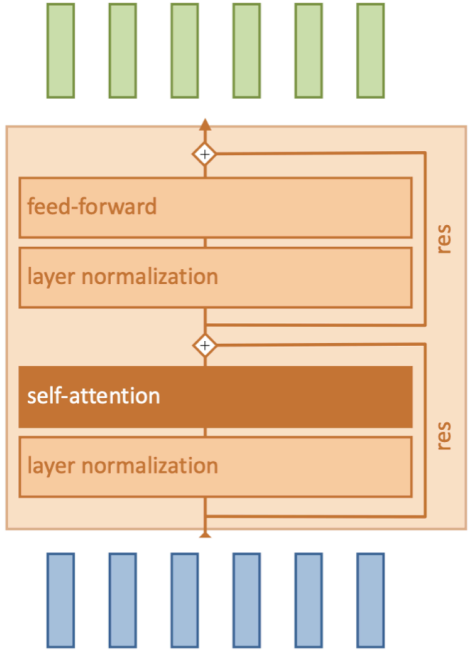
- Project input to lower-dimensional embeddings
- Stack them so we can feed them through self-attention at once
- Unstack and project back to original dimensions

```
qkv = nn.Linear(input_dim, 3*embed_dim)(x) # project to embed_dim
qkv = qkv.reshape(batch_size, seq_length, num_heads, 3*head_dim)
q, k, v = qkv.chunk(3, dim=-1)

values, attention = scaled_dot_product(q, k, v, mask=mask) # self-att
values = values.reshape(batch_size, seq_length, embed_dim)
out = nn.Linear(embed_dim, input_dim) # project back
```

Attention block

The attention block is quite straightforward



Attention block

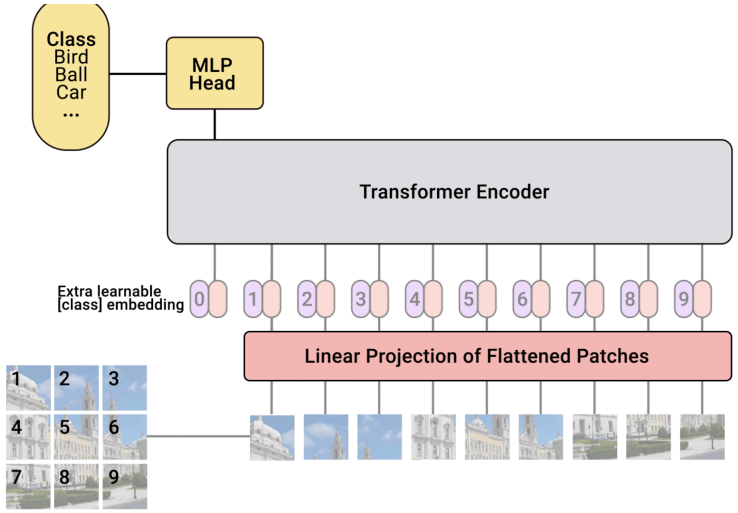
```
def __init__(self, embed_dim, hidden_dim, num_heads, dropout=0.0):
    self.layer_norm_1 = nn.LayerNorm(embed_dim)
    self.attn = nn.MultiheadAttention(embed_dim, num_heads)
    self.layer_norm_2 = nn.LayerNorm(embed_dim)
    self.linear = nn.Sequential( # Feed-forward layer
        nn.Linear(embed_dim, hidden_dim),
        nn.GELU(), nn.Dropout(dropout),
        nn.Linear(hidden_dim, embed_dim),
        nn.Dropout(dropout)
    )
```

```
def forward(self, x):
    inp_x = self.layer_norm_1(x)
    x = x + self.attn(inp_x, inp_x, inp_x)[0] # self-att + res
    x = x + self.linear(self.layer_norm_2(x)) # feed-fw + res
    return x
```

Vision transformer

Final steps:

- Linear projection (embedding) to map patches to vector
- Add classification token to input
- 2D positional encoding
- Small MLP head to map CLS token to prediction




```
GPU available: True (mps), used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
Lightning automatically upgraded your loaded checkpoint from v1.6.4 to v
2.5.0.post0. To apply the upgrade to your files permanently, run `python
-m pytorch_lightning.utilities.upgrade_checkpoint ../data/checkpoints/Vi
T.ckpt`
```

Summary

- Tokenization
    - Find a good way to split data into tokens
- Word/Image embeddings (for initial embeddings)
    - For text: Word2Vec, FastText, GloVe
    - For images: MLP, CNN,...
- Sequence-to-sequence models
    - 1D convolutional nets (fast, limited memory)
    - RNNs (slow, also quite limited memory)
- Transformers
    - Self-attention (allows very large memory)
    - Positional encoding
    - Autoregressive models
- Vision transformers
    - Useful if you have lots of data (and compute)

Acknowledgement

Several figures came from the excellent VU Deep Learning course.