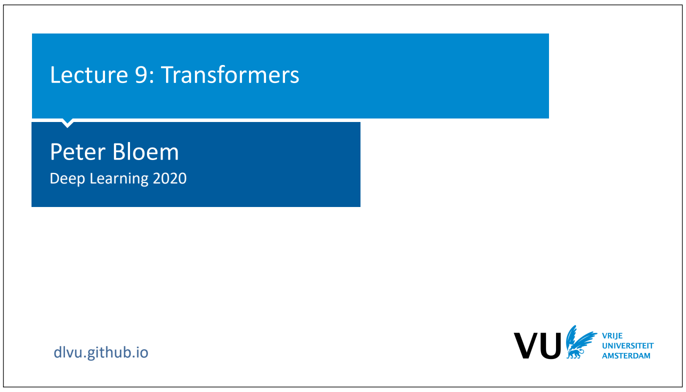


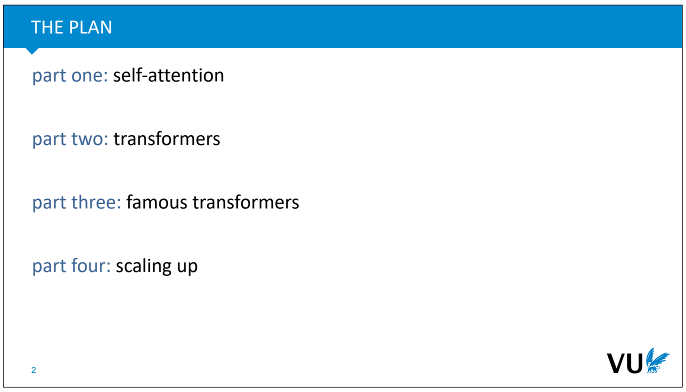
Lecture 9: Transformers

Peter Bloem
Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

In a previous lecture, we looked at sequence models, and we saw two examples: RNNs and Convolutions. In this lecture, we will add a third: **self-attention**. This is the basic building block of the **transformer** family of architectures. This is the type or architecture behind modern chatbots like ChatGPT, Bard and Claude.



THE PLAN

part one: self-attention

part two: transformers

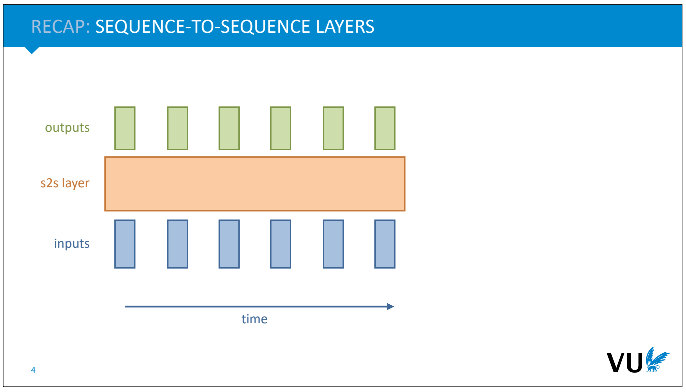
part three: famous transformers

part four: scaling up

PART ONE: SELF-ATTENTION

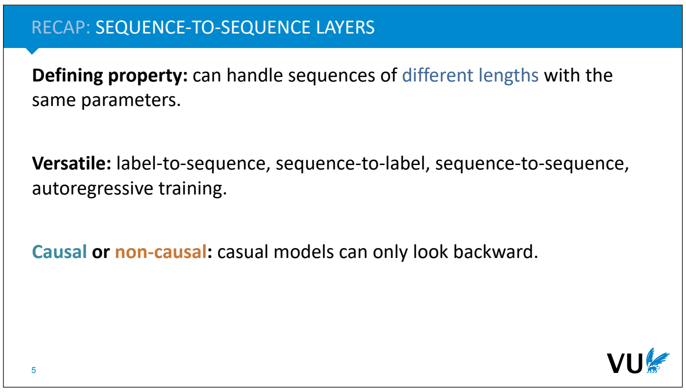
|section|Self-attention|

|video|https://www.youtube.com/embed/KmAISyVvE1Y?si=iTI_8duI_vIbCS3u|



RECAP: SEQUENCE-TO-SEQUENCE LAYERS

Here is a recap of the main point to remember from the lecture on sequence models. The sequence-to-sequence layer is an abstract class of layer. It consumes a sequence of vectors, and produces as output a sequence of vectors. The sequences have the same length. The dimension of the vectors doesn't need to be the same.



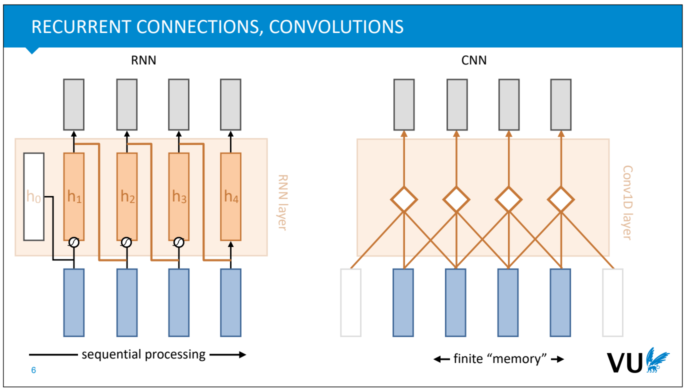
RECAP: SEQUENCE-TO-SEQUENCE LAYERS

Defining property: can handle sequences of different lengths with the same parameters.

Versatile: label-to-sequence, sequence-to-label, sequence-to-sequence, autoregressive training.

Causal or non-causal: casual models can only look backward.

The key property of sequence-to-sequence layers is that they can be applied to sequences of different lengths, *with the same set of weights*. This is what makes it difficult to find mechanisms for sequence-to-sequence layers, but also what makes them versatile.



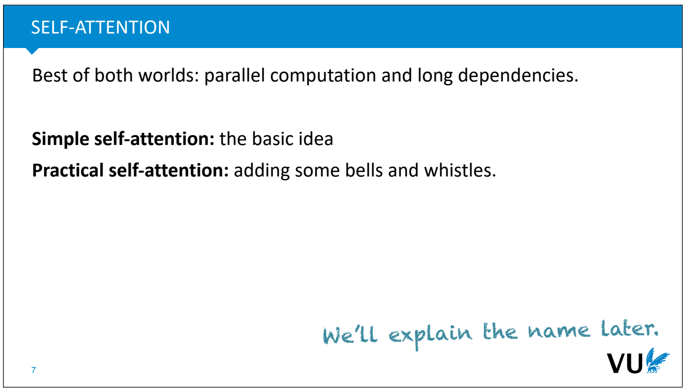
RECURRENT CONNECTIONS, CONVOLUTIONS

We've seen two examples of (non-trivial) sequence-to-sequence layers so far: recurrent neural networks, and convolutions. RNNs have the benefit that they can potentially look infinitely far back into the sequence, but they require fundamentally sequential processing, making them slow.

This is shown by the orange connections: for the computation of h_2 *to start, we need the computation of* h_1 *to have finished.*

Convolution don't have this drawback—we can compute each output vector in parallel if we want to—but the downside of convolutions is that they are limited in how far back they can look into the sequence. The output all the way on the right, cannot depend in the input all the way on the left.

Self-attention is another sequence-to-sequence layer, and one which provides us with the best of both worlds: parallel processing and a potentially infinite memory.



SELF-ATTENTION

Best of both worlds: parallel computation and long dependencies.

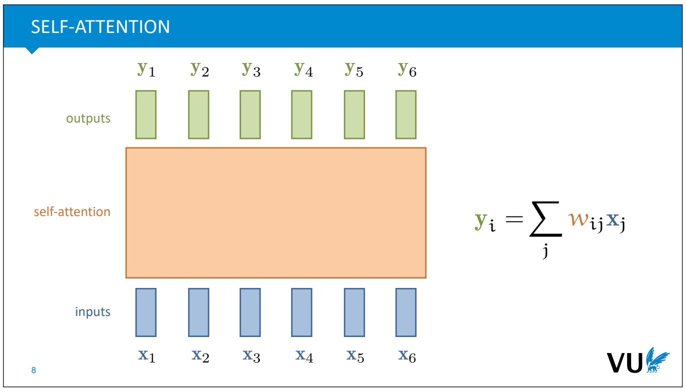
Simple self-attention: the basic idea

Practical self-attention: adding some bells and whistles.

We'll explain the name later.

We will explain self-attention in two steps. First, a version that is stripped down to the basic mechanism. This lacks a few bells and whistles, but it already provides us with a functioning sequence to sequence layer.

Then, we will add a few features that are commonly used to make the method a bit more versatile and powerful.



SELF-ATTENTION

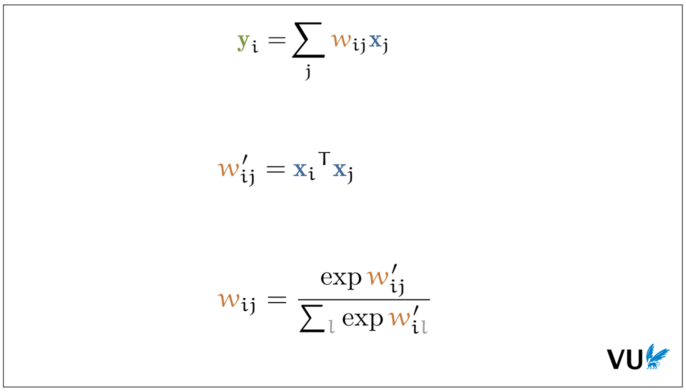
y_1 y_2 y_3 y_4 y_5 y_6
outputs
self-attention
inputs
x_1 x_2 x_3 x_4 x_5 x_6

$$y_i = \sum_j w_{ij} x_j$$

At heart, the operation of self-attention is very simple. Every output is simply a *weighted sum* over the inputs. For a given output y_i, we multiply each input vector x_j with a scalar w_{ij}, Then we sum up all the vectors, and the result is y_i.

The trick is that the weights in this sum are *not parameters*. They are *derived* from the inputs.

Note that this means that the input and output dimensions of a self-attention layer are always the same. If we want to transform to a different dimension, we'll need to add a projection layer after the self-attention.



$$\mathbf{y}_i = \sum_j w_{ij}\mathbf{x}_j$$

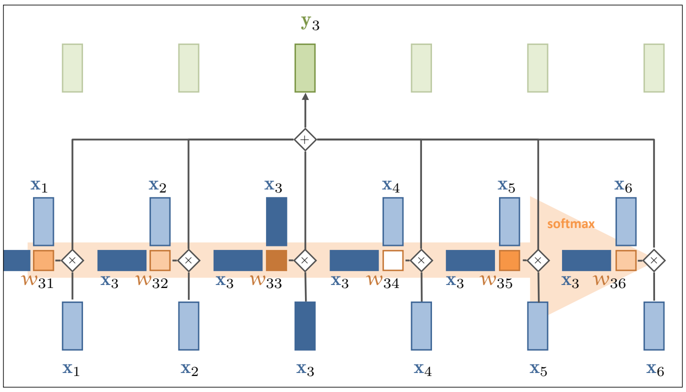
$$w'_{ij} = {\mathbf{x}_i}^\mathsf{T}\mathbf{x}_j$$

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_l \exp w'_{il}}$$

For each combination of an input j and an output i, we derive the weight by taking the dot product of the two inputs $\mathbf{x}_i$ and $\mathbf{x}_j$. That is the input vector, and the input vector in the same position as the current output vector.

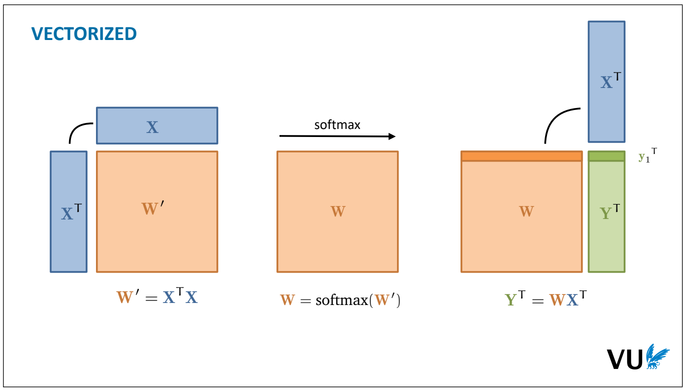
Their dot product forms the raw weight. Roughly speaking, this means that the more similar $\mathbf{x}_i$ and $\mathbf{x}_j$ are, the higher we weigh $\mathbf{x}_i$ in the weighted sum that we return.

Then, we would like all the weight to sum to one, for a given output. To achieve this, we apply a softmax over all weights that correspond to a single output $\mathbf{y}_j$.



Here is the whole process in an animation for one vector in the output sequence.

Note that this does indeed provide a sequence-to-sequence operation. If the next sequence is longer or shorter, we can apply exactly the same operation. In fact, we haven't yet introduced any parameters.



VECTORIZED

$\mathbf{W}' = \mathbf{X}^\mathsf{T}\mathbf{X}$

$\mathbf{W} = \text{softmax}(\mathbf{W}')$

$\mathbf{Y}^\mathsf{T} = \mathbf{W}\mathbf{X}^\mathsf{T}$

Self attention looks particularly simple when vectorized, that is, expressed terms of matrix operations.

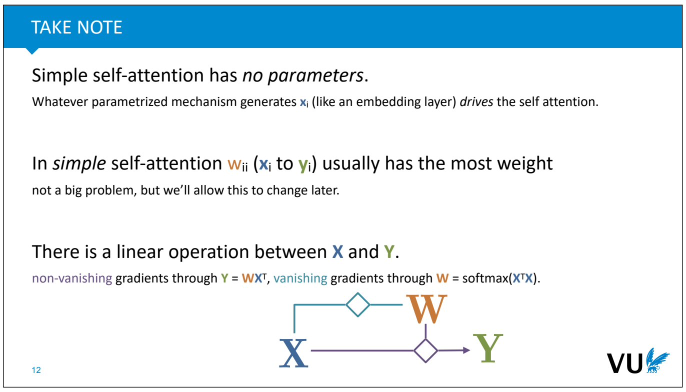
Note first, that if we arrange all the weights for one output vector, say y1, in a long vector **w**, with one element for every input, then the computation of the weighted sum for y1, is just w (as a row vector) times $\mathbf{X}^\mathsf{T}$, where X is a matrix containing all the input vectors. If we then do the same for the other output vectors, and concatenate all the w's into a matrix W as its rows, we see that the multiplication $\mathbf{WX}^\mathsf{T}$ computes a matrix **Y** with all output vectors as its columns.

Next, we need to figure out how to easily compute this matrix of weights. We'll start with a matrix of raw weights W'. The element ij of this matrix is the dot product of two vectors i and j. This is essentially the definition of a matrix multiplication: in the multiplication **AB** = **C**, $\mathbf{C}_{ij}$ is the dot product of the i-th row of **A** and the j-th column of **B**. This means that if we set up on matrix with $\mathbf{x}_i$ as its i-th row and one matrix with $\mathbf{x}_i$ as its j-th column, multiplying these together will give us **W'**.

Finally, we need to apply a softmax. This should go over all the weights that correspond to a single output. In the multiplication diagram on the right, we see that these form the *rows* of **W'**. Therefore, we need to apply a row-wise softmax.

Put differently, we want to ensure that all the rows sum to 1.

With that, we have the three steps of a vectorized, simple self-attention.



TAKE NOTE

Simple self-attention has *no parameters*.

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

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.

There is a linear operation between **X** and **Y**.

non-vanishing gradients through $Y = WX^T$, vanishing gradients through $W = \text{softmax}(X^TX)$.

12

There are some important properties of this mechanism to note.

First, notice that we haven't yet added any parameters. The whole mechanism is defined by whatever the input to the self attention is.

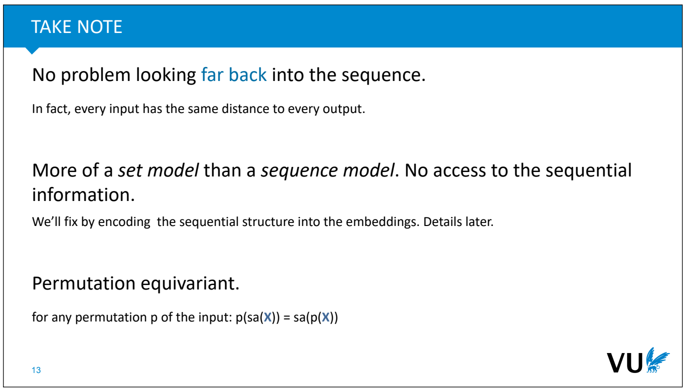
It's important to realize that this doesn't stop us from learning. For instance, we can feed the a single, simple self-attention with a sequence of embedding layers. This would already result in a model that is model flexible than just the embeddings by themselves (since there is a small amount of interaction between the embeddings added by the self-attention).

Next, note that the dot product of a vector with itself is usually bigger than that of two other vectors. This means that in the weighted sum for output i, input i will have the biggest contribution. In fact, we can think of the simple self attention as mostly reproducing the input, but mixing into each vector a little bit of the content of the other vectors in the sequence.

If we were to stack many simple self-attentions together, they would probably end up with a uniform mixture over all inputs.

Finally, one of the benefits of the self attention becomes clear if we draw its computation graph as shown in the slides. If we forget where W came from, the whole operation become entirely *linear*. This means it's easy to learn, and it provides strong gradients upstream. The nonlinearity is confined to a second computation path, over which we'll get poorer gradients, but ultimately a more expressive operation.

Compare this to the residual connection. There too, we provide one linear path for simple, strong gradients early on, and a second computation path, for more complex gradients, once learning gets underway.



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))

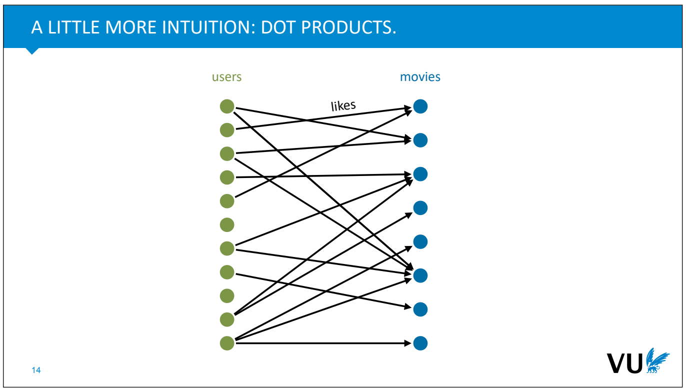
Here are some more properties that are important.

Unlike the RNN, there is no difference between nearby inputs and far away inputs. They all influence the output in the same way. One way of saying this is that the distance on the computation graph from one output is the same to every input.

In fact, the simple self attention doesn't have access to the sequential structure of the input.

We can say that the simple self-attention is *permutation equivariant*: if we shuffle the input over the time dimension, the resulting output is shuffled in the same way. Equivalently shuffling the input and applying the self-attention gives the same result as applying the self attention and shuffling.

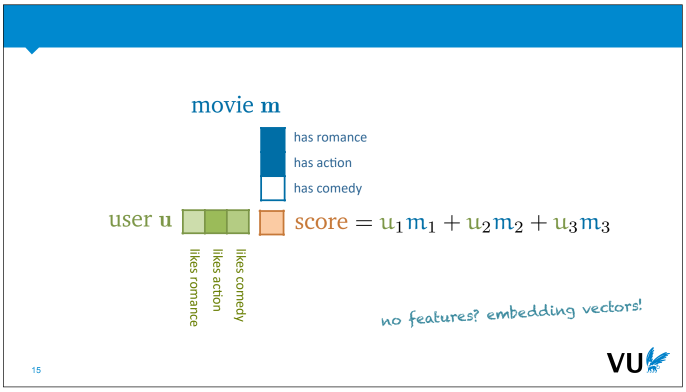
These are essentially three different ways of saying the same thing.



A LITTLE MORE INTUITION: DOT PRODUCTS.

To build some intiuition for *why* the self attention works, how it allows us to learn the sort of thing that we want, we need to look into how **dot products** function. To do so, we'll leave the realm of sequence learning for a while and dip our toes briefly into the waters of *recommendation*.

Imagine that we have a set of users and a set of movies, with no features about any of them except an incomplete list of which user liked which movie. Our task is to predict which other movies a given user will like.



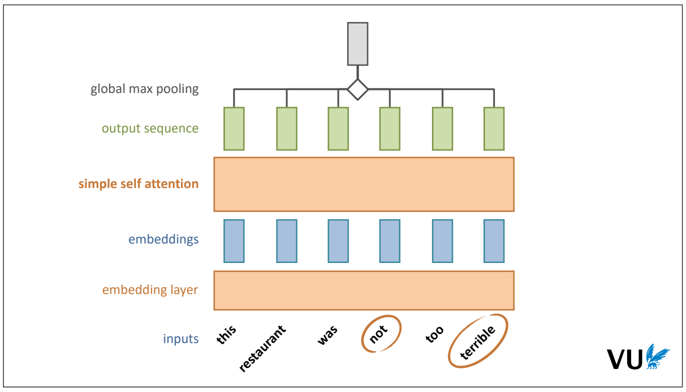
$\text{score} = u_1 m_1 + u_2 m_2 + u_3 m_3$

no features? embedding vectors!

If we had features for each movie and user, we could match them up like this. We multiply how much the user likes romance by how much romance there is in the movie. If both are positive of negative, the score is increased. If one is positive and one is negative, the score is decreased.

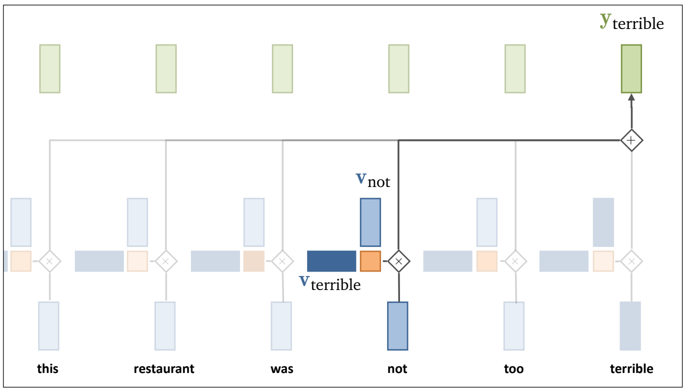
Note that we're not just taking into account the sign of the values, but also the magnitude. If a user's preference for action is near zero, it doesn't matter much for the score whether the movie has action.

How do we translate this back to self attention? There these scores are our *weights*. If we think of the input vectors as containing information, and the weight w_{ij} as a score for how important input i is for output j, we get a sense of how the self attention operates (we can clarify this better when we add some bells and whistles).



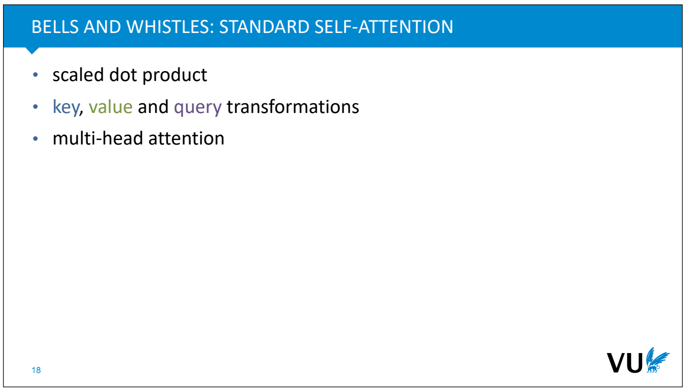
As a simple example of how you might use self-attention, let's build a sequence classifier consisting of just one embedding layer followed by a global maxpooling layer. We'll imagine a sentiment classification task where the aim is to predict whether a restaurant review is positive or negative.

If we did this without the self-attention layer, we would essentially have a model where each word can only contribute to the output score *independently* of the other. This is known as **a bag-of-words model**. In this case, the word "terrible" would probably cause us to predict that this is a negative review. In order to see that it might actually be a positive review, we need to recognize that the meaning of the word terrible is moderated by the word not. This is what the self-attention can do for us.


$\mathbf{y}_{\text{terrible}}$
$\mathbf{v}_{\text{not}}$
$\mathbf{v}_{\text{terrible}}$
this
restaurant
was
not
too
terrible

If the embedding vectors of *not* and *terrible* have a high dot product together, the weight of the input vector for not becomes high, allowing it to influence the meaning of the word terrible in the output sequence.

The self attention still can't see that the words are close together, but our model might already be able to learn that if the word not *appears in a sentence, it should be a little bit more cautious about how to interpret the word* terrible.

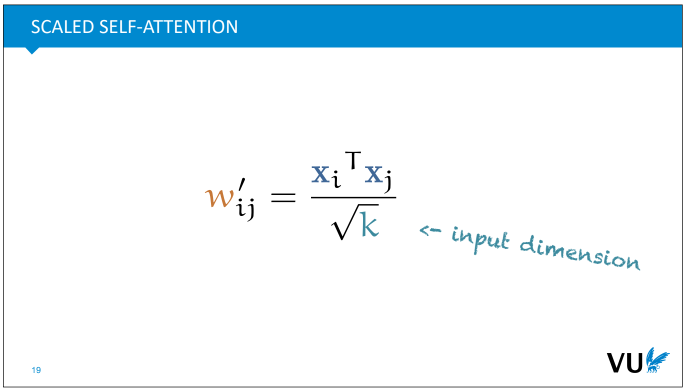


BELLS AND WHISTLES: STANDARD SELF-ATTENTION

- scaled dot product
- key, value and query transformations
- multi-head attention

18

The standard self attention add some bells and whistles to this basic framework. We'll discuss the three most important additions.



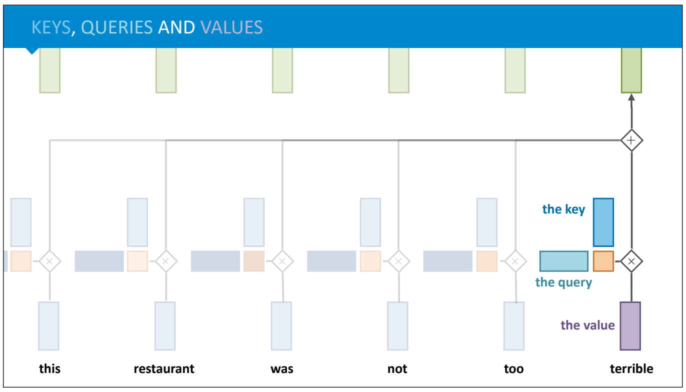
SCALED SELF-ATTENTION

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

<- input dimension

Scaled self attention is very simple: instead of using the dot product, we use the dot product scaled by the square root of the input dimension. This ensures that the input and output of the self attention operation have similar variance.

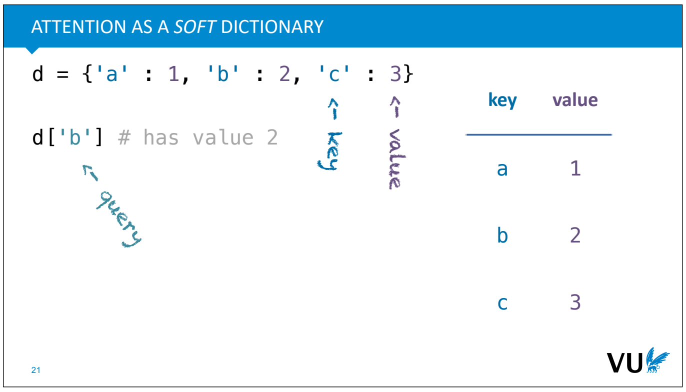
Why √k? Imagine a vector in $\mathbb{R}^k$ with values all c. Its Euclidean length is √kc. Therefore, we are dividing out the amount by which the increase in dimension increases the length of the average vectors. Transformer usually models apply normalization at every layer, so we can usually assume that the input is standard-normally distributed.



KEYS, QUERIES AND VALUES

In each self attention computation, every input vector occurs in three distinct roles:

- **the value**: the vector that is used in the weighted sum that ultimately provides the output
- **the query**: the input vector that corresponds to the current output, matched against every other input vector.
- **the key**: the input vector that the query is matched against to determine the weight.



ATTENTION AS A *SOFT* DICTIONARY

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

d['b'] # has value 2
```

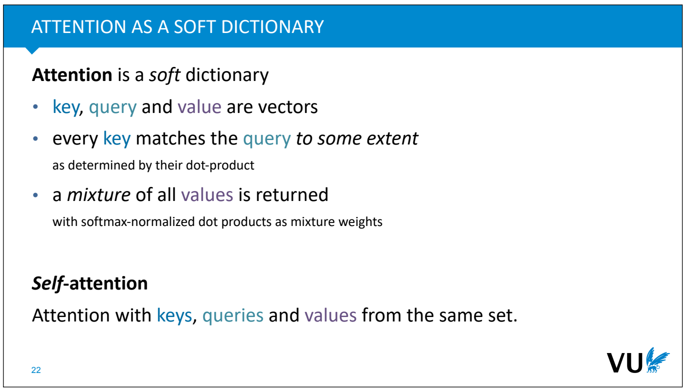
<- key

<- value

<- query

key	value
a	1
b	2
c	3

In a dictionary, all the operations are discrete: a query only matches a single key, and returns only the value corresponding to that key.



ATTENTION AS A SOFT DICTIONARY

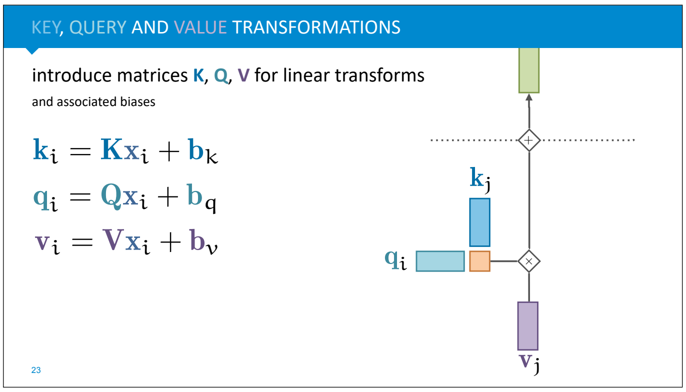
Attention is a *soft* dictionary

- 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.

If the dot product of only one query/key pair is non-zero, we recover the operation of a normal dictionary.



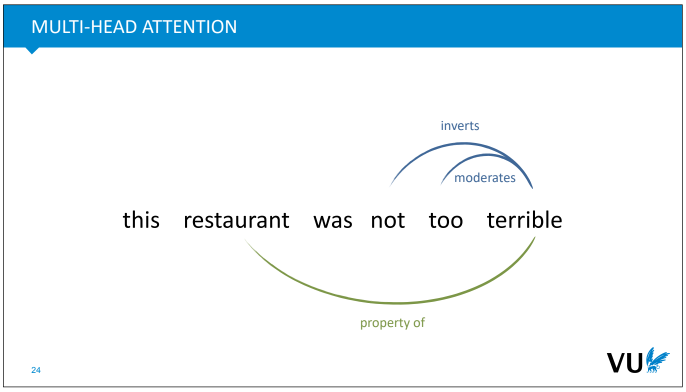
KEY, QUERY AND VALUE TRANSFORMATIONS

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

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

To give the self attention some more flexibility in determining its behavior, we multiply each input vector by three different k-by-k parameter matrices, which gives us a different vector to act as key query and value.

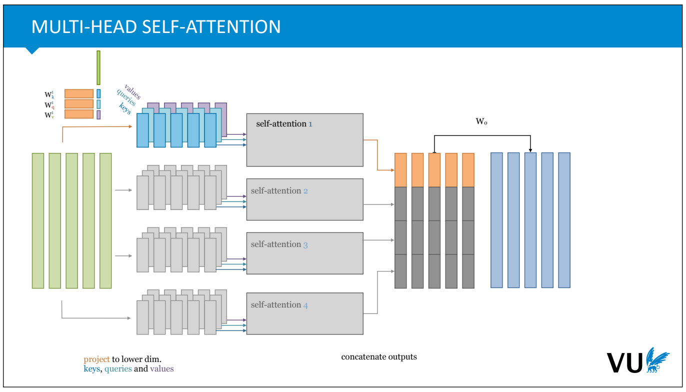
Note that this makes the self attention operation a layer with parameters (where before it had none).



MULTI-HEAD ATTENTION

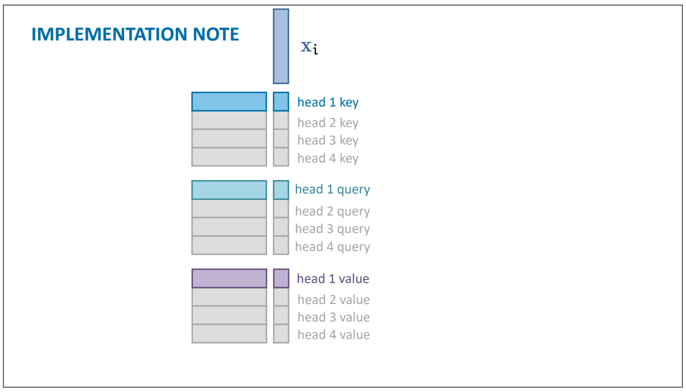
In many sentences, there are different relations to model. Here, the word meaning of the word “terrible” is inverted by “not” and moderated by “too”. Its relation to the word restaurant is completely different: it describes a property of the restaurant.

The idea behind multi-head self-attention is that multiple relations are best captured by different self-attention operations.



MULTI-HEAD SELF-ATTENTION

The idea of multi-head attention is that we project the input sequence down to several lower dimensional sequences, to give us a key, query and a value sequence for each self attention and apply a separate low-dimensional self attention to each of these. After this, we concatenate their outputs, and apply another linear transformation (biases not shown)



IMPLEMENTATION NOTE

$\mathbf{x}_i$

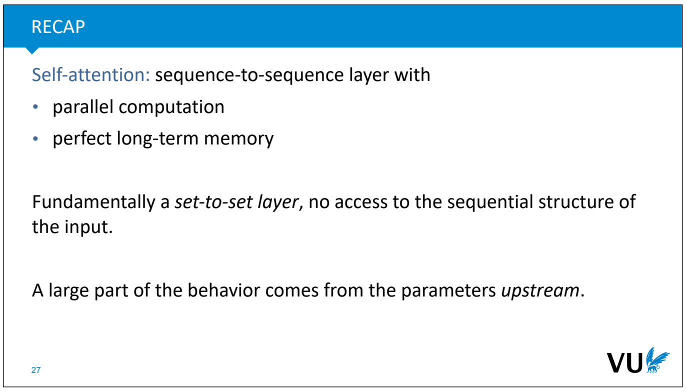
head 1 key
head 2 key
head 3 key
head 4 key

head 1 query
head 2 query
head 3 query
head 4 query

head 1 value
head 2 value
head 3 value
head 4 value

Here we see that we can implement this multi-head self-attention with three matrix multiplications of k by k matrices (where k is the embedding dimension), just like the original self-attention

NB. the matrix multiplication by W^o after concatenation is an addition. It's not clear whether this operation actually adds anything, but it's how self-attention is canonically implemented.



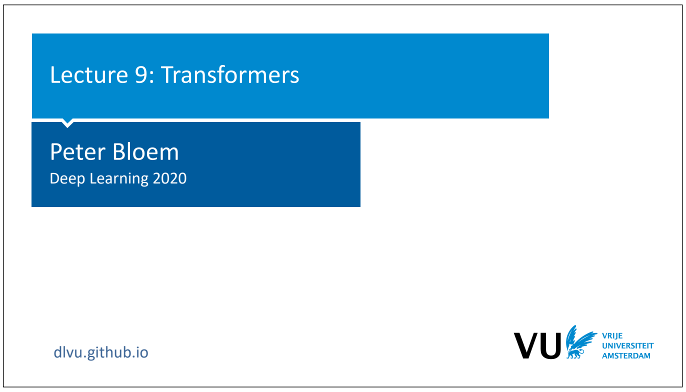
RECAP

Self-attention: sequence-to-sequence layer with

- parallel computation
- perfect long-term memory

Fundamentally a *set-to-set layer*, no access to the sequential structure of the input.

A large part of the behavior comes from the parameters *upstream*.

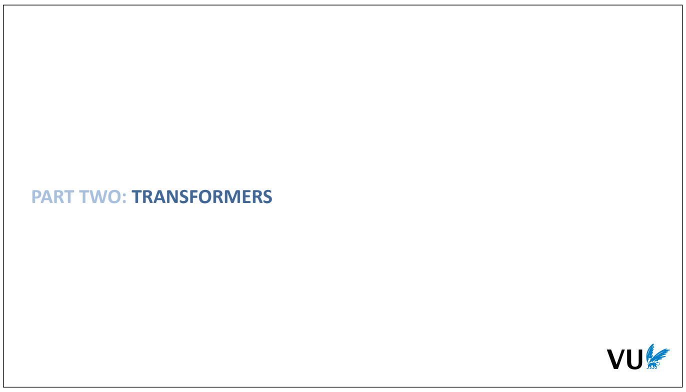


Lecture 9: Transformers

Peter Bloem

Deep Learning 2020

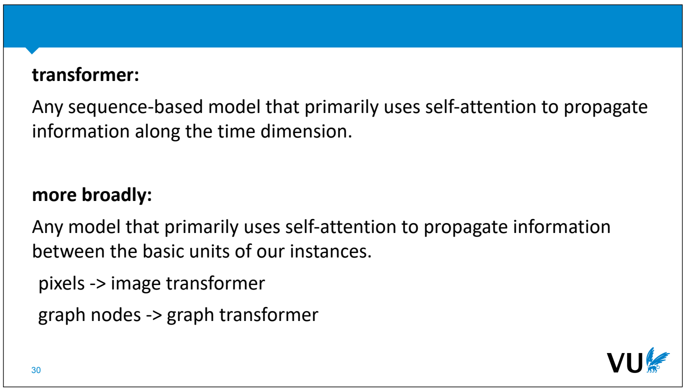
dlvu.github.io



PART TWO: TRANSFORMERS

|section|Transformers|

|video|https://www.youtube.com/embed/oUhGZMCTHtI?si=50SqAJ9u9uyArZKB|



transformer:

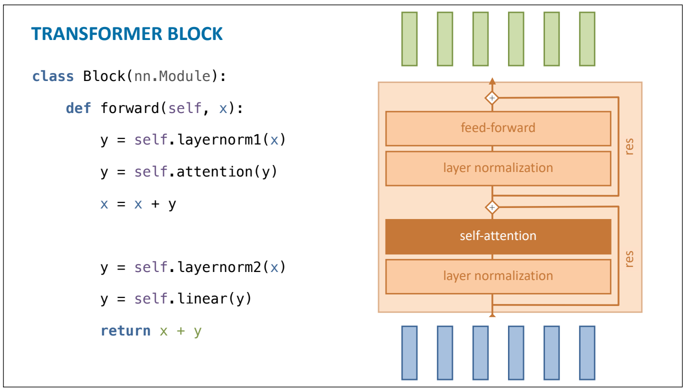
Any sequence-based model that primarily uses self-attention to propagate information along the time dimension.

more broadly:

Any model that primarily uses self-attention to propagate information between the basic units of our instances.

pixels -> image transformer

graph nodes -> graph transformer



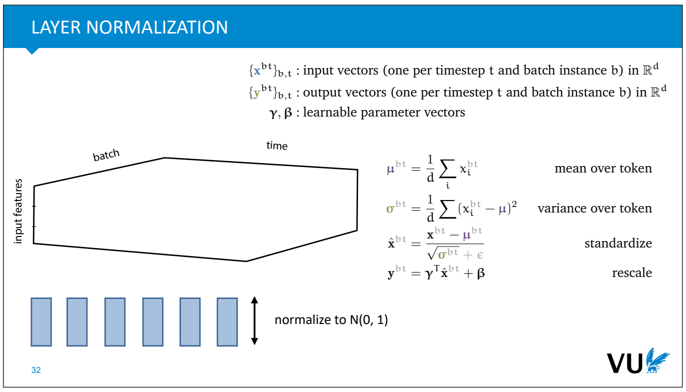
TRANSFORMER BLOCK

```
class Block(nn.Module):

    def forward(self, x):
        y = self.layernorm1(x)
        y = self.attention(y)
        x = x + y

        y = self.layernorm2(x)
        y = self.linear(y)
        return x + y
```

The basic building block of transformer models is usually a simple **transformer block**.

The details differ per transformer, but the basic ingredients are usually: one self-attention, one feed-forward layer applied individually to each token in the sequence and a layer normalization and residual connection for each.

Note that the self-attention is the only operation in the block that propagates information across the time dimension. The other layers operate only on each token independently.



LAYER NORMALIZATION

$\{\mathbf{x}^{bt}\}_{b,t}$: input vectors (one per timestep t and batch instance b) in $\mathbb{R}^d$

$\{\mathbf{y}^{bt}\}_{b,t}$: output vectors (one per timestep t and batch instance b) in $\mathbb{R}^d$

$\boldsymbol{\gamma}, \boldsymbol{\beta}$: learnable parameter vectors

$$\mu^{bt} = \frac{1}{d}\sum_i x_i^{bt}$$ mean over token

$$\sigma^{bt} = \frac{1}{d}\sum (x_i^{bt} - \mu)^2$$ variance over token

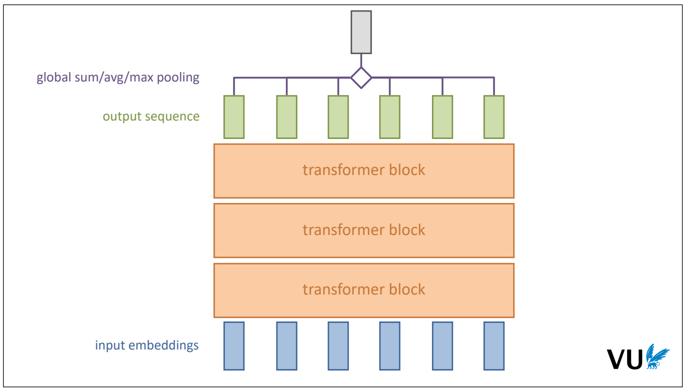
$$\hat{\mathbf{x}}^{bt} = \frac{\mathbf{x}^{bt} - \mu^{bt}}{\sqrt{\sigma^{bt}} + \epsilon}$$ standardize

$$\mathbf{y}^{bt} = \boldsymbol{\gamma}^T \hat{\mathbf{x}}^{bt} + \boldsymbol{\beta}$$ rescale

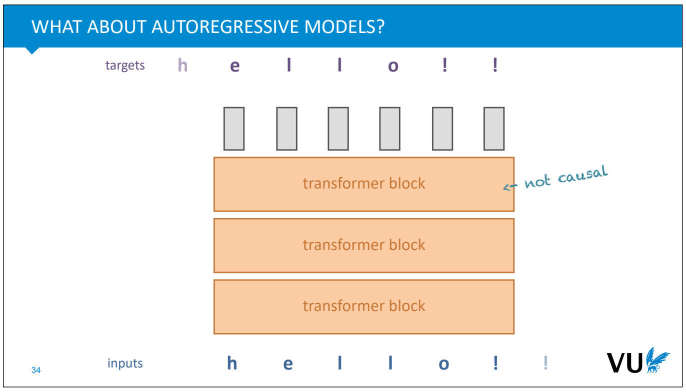
Layer normalization is like batch normalization, except that it normalizes along a different dimension of the batch tensor.

Note that this does not propagate information across the time dimension. That is still reserved for the self attention only.

While layer normalization tends to work a little less well than batch normalization, the great benefit here is that its behavior doesn't depend on the batch size. This is important, because transformer models are often so big that we can only train on single-instance batches. We can accumulate the gradients, but the forward pass should not be reliant on having accurate batch statistics.



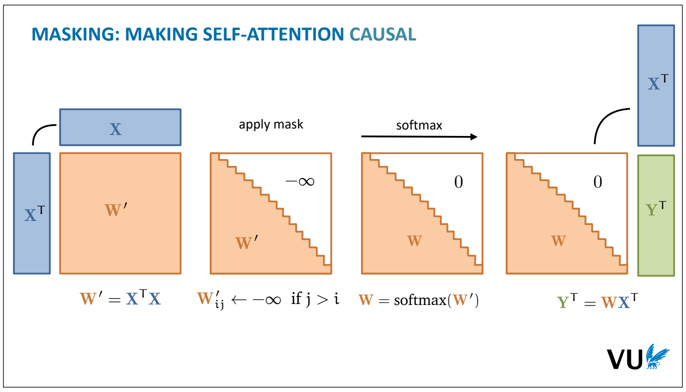
Once we've defined a transformer block, all we need to do is stack a bunch of them together. Then, if we have a sequence-to-label task, we just need one global pooling operation and we have a sequence-to-label model.



WHAT ABOUT AUTOREGRESSIVE MODELS?

What about autoregressive modeling?

If we do this naively, we have a problem: the self-attention operation can just look ahead in the sequence to predict what the next model will be. We will never learn to predict the future from the past. In short the transformer block is not a *causal* sequence-to-sequence operation.

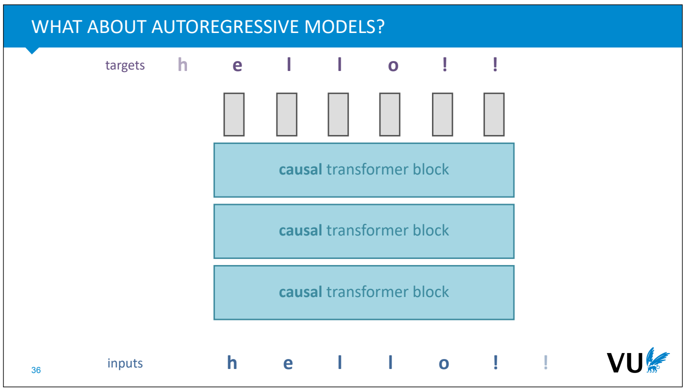


MASKING: MAKING SELF-ATTENTION CAUSAL

$W' = X^TX$ $\quad W'_{ij} \leftarrow -\infty \text{ if } j > i$ $\quad W = \text{softmax}(W')$ $\quad Y^T = WX^T$

The solution is simple: when we compute the attention weights, we mask out any attention from the current token to future tokens in the sequence.

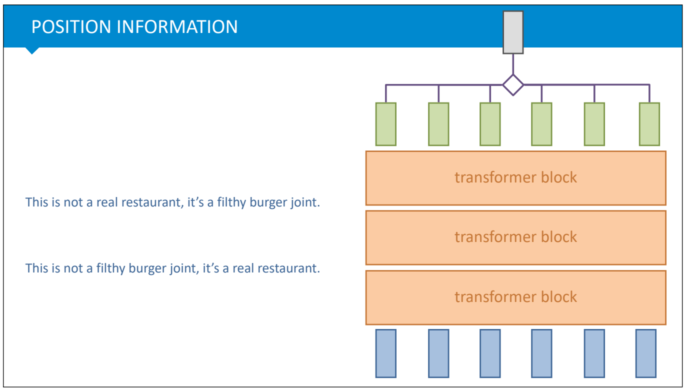
Note that to do this, we need to set the raw attention weights to negative infinity, so that after the softmax operation, they become 0.



WHAT ABOUT AUTOREGRESSIVE MODELS?

Since the self attention is the only part of the transformer block that propagates information across the time dimension, making that part causal, makes the whole block causal.

With a stack of causal transformer blocks, we can easily build an autoregressive model.



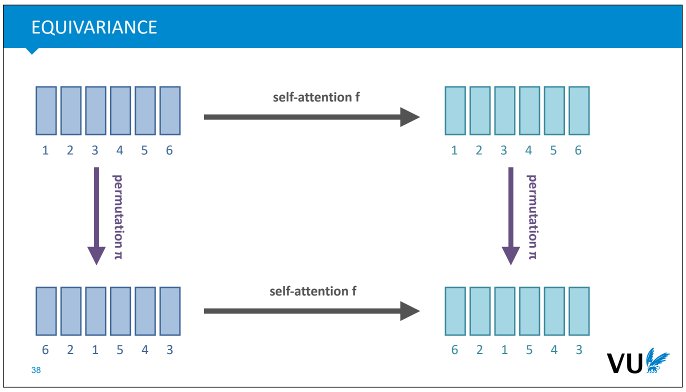
POSITION INFORMATION

This is not a real restaurant, it's a filthy burger joint.

This is not a filthy burger joint, it's a real restaurant.

To really interpret the meaning of the sentence, we need to be able to access the position of the words. Two sentences with their words shuffled can mean the exact opposite thing.

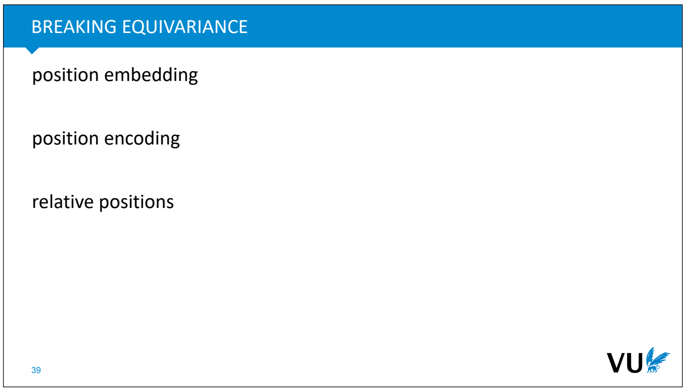
If we feed these sentences, tokenized by word, to the architecture on the right, their output label will necessarily be the same. The self-attention produces the same output vectors, with just the order differing in the same way they do for the two inputs, and the global pooling just sums all the vectors irrespective of position.



EQUIVARIANCE

This is a property known as **equivariance**. Self-attention is *permutation* equivariant. Whether we permute the tokens in the sequence first and then apply self-attention, or apply self attention and then permute, we get the same result. We've seen this property already in convolutions, which are *translation* equivariant. This tells us that equivariance is not a bad thing; it's a property that allows us to control what structural properties the model assumes about the data.

Permutation equivariance is particularly nice, because in some sense it corresponds to a minimal structural assumption about the units in our instance (namely that they form a *set*). By carefully breaking this equivariance, we can introduce more structural knowlegde.



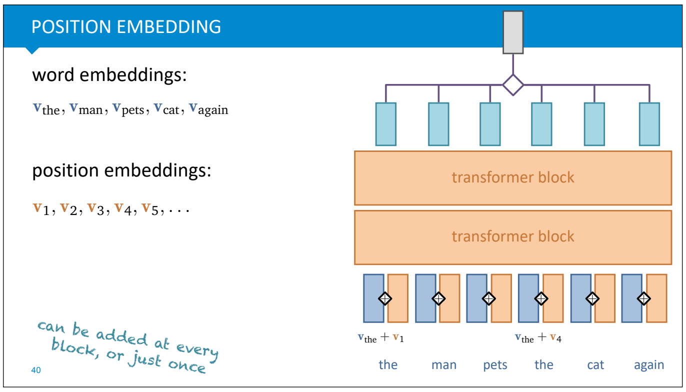
BREAKING EQUIVARIANCE

position embedding

position encoding

relative positions

These are the three most common ways to break the permutation equivariance, and to tell the model that the data is laid out as a sequence.



POSITION EMBEDDING

word embeddings:

$\mathbf{v}_{\text{the}}, \mathbf{v}_{\text{man}}, \mathbf{v}_{\text{pets}}, \mathbf{v}_{\text{cat}}, \mathbf{v}_{\text{again}}$

position embeddings:

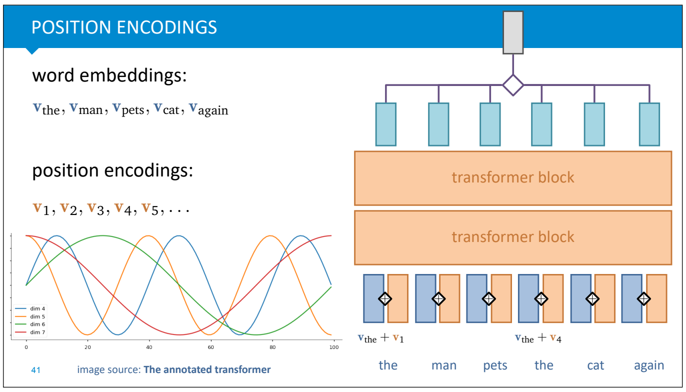
$\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5, \ldots$

transformer block
transformer block
$\mathbf{v}_{\text{the}} + \mathbf{v}_1$
$\mathbf{v}_{\text{the}} + \mathbf{v}_4$
the
man
pets
the
cat
again

can be added at every block, or just once

The idea behind position embeddings is simple. Just like we assign each word in our vocabulary an embedding vector, we also assign each *position* in our vocabulary an embedding vector. This way, the input vectors for the first “the” in the input sequence and the second “the” are different, because the first is added to the position embedding $\mathbf{v}_1$ and the second is added to the input embedding $\mathbf{v}_2$.

This break our equivariance: the position information becomes *part* of our embedding vectors, and is fed into the self attention. This is very effective, and very easy to implement. The only drawback is that we can’t run the model very well on sequences that are longer than the largest position embedding observed during training.



POSITION ENCODINGS

word embeddings:

$\mathbf{v}_{\text{the}}, \mathbf{v}_{\text{man}}, \mathbf{v}_{\text{pets}}, \mathbf{v}_{\text{cat}}, \mathbf{v}_{\text{again}}$

position encodings:

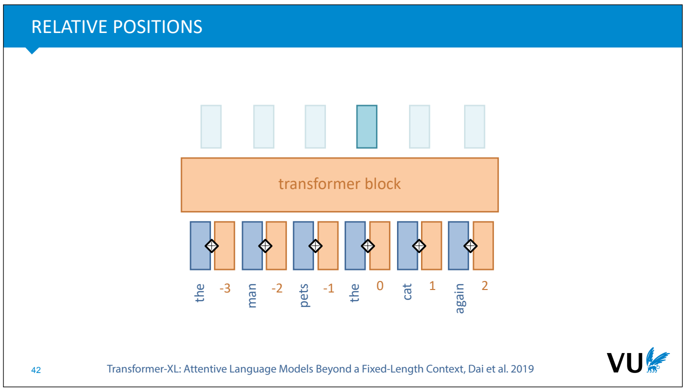
$\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5, \ldots$

image source: The annotated transformer

Position encodings are very similar. Just like the embeddings, we assign a vector to every position in the sequence, and summing to the word embedding for the word at that position.

The difference is that the position encodings are *not learned*. They are fixed to some function that we expect the downstream self-attentions can easy latch on to to tell the different positions apart. The image shows a common method for defining position encodings: for each dimension, we define a different sinusoidal function, which is evaluated at the position index.

The main benefit is that this pattern is predictable, so the transformer can theoretically model it. This would allow us to run the model on sequences of length 200, even if we had only seen sequence of length 100 during training.



RELATIVE POSITIONS

Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context, Dai et al. 2019

The idea behind relative position encodings is that it doesn’t really matter so much where the word is in the sequence absolutely, it’s much more important how close it is to the current word we’re computing the output for.

Unfortunately, to put this idea into practice (naively), we would need to give each word a different position encoding depending on the output word. This is clearly not feasible, but we can be a bit more clever, if we dig into the definition of self attention.

RELATIVE POSITIONS

$$\sqrt{d}w'_{ij} = q_i^{\mathsf{T}} k_j = (Qx_i)^{\mathsf{T}} Kx_j = x_i^{\mathsf{T}} Q^{\mathsf{T}} K x_j$$

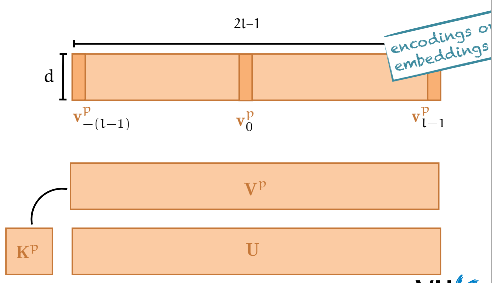
$$= (v_i^w + v_i^p)^{\mathsf{T}} Q^{\mathsf{T}} K (v_j^w + v_j^p)$$

$$\begin{aligned} = \; & v_i^{w\mathsf{T}} Q^{\mathsf{T}} K v_j^w \\ & + v_i^{w\mathsf{T}} Q^{\mathsf{T}} K v_j^p \\ & + v_i^{p\mathsf{T}} Q^{\mathsf{T}} K v_j^w \\ & + v_i^{p\mathsf{T}} Q^{\mathsf{T}} K v_j^p \end{aligned}$$

$$\begin{aligned} \sqrt{d}w'_{ij} = \; & v_i^{w\mathsf{T}} Q^{\mathsf{T}} K^w v_j^w \\ & + v_i^{w\mathsf{T}} Q^{\mathsf{T}} K^p v_{i-j}^p \\ & + a^{\mathsf{T}} K^w v_j^w \\ & + b^{\mathsf{T}} K^p v_{i-j}^p \end{aligned}$$

COMPUTING RELATIVE POSITION ENCODINGS/EMBEDDINGS

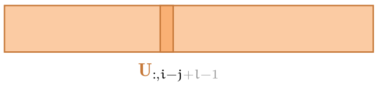
$v_i^{w\mathsf{T}} Q^{\mathsf{T}} K^p v_{i-j}^p$

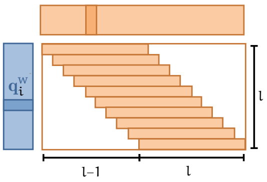


$U = K^p V^p$

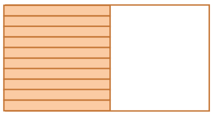
COMPUTING RELATIVE POSITION ENCODINGS

$v_i^{w\mathsf{T}} Q^{\mathsf{T}} K^p v_{i-j}^p = q_i^{w\mathsf{T}} U_{:,i-j+l-1}$



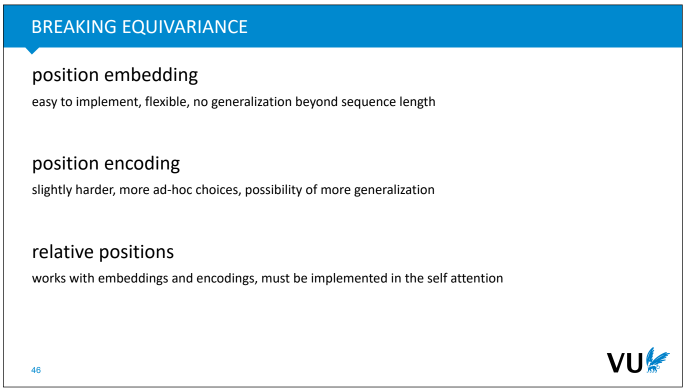


```
qu = qu.view(-1)
# size l(2l − 1)

pad qu with l 0s
# size 2*l*l

qu = qu.view(l, 2*l)
```



```
qu = qu[:, :l]
```

Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context, Dai et al. 2019
Appendix B



BREAKING EQUIVARIANCE

position embedding

easy to implement, flexible, no generalization beyond sequence length

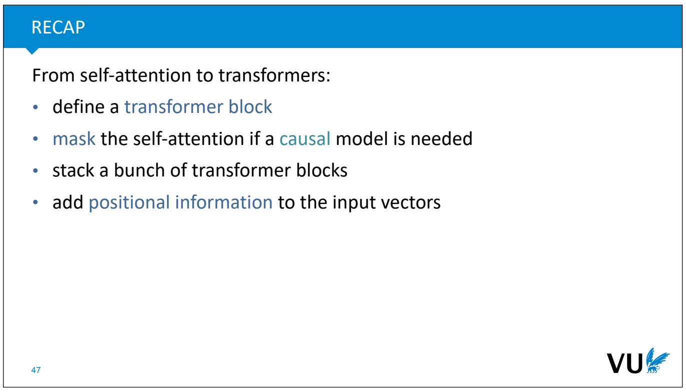
position encoding

slightly harder, more ad-hoc choices, possibility of more generalization

relative positions

works with embeddings and encodings, must be implemented in the self attention

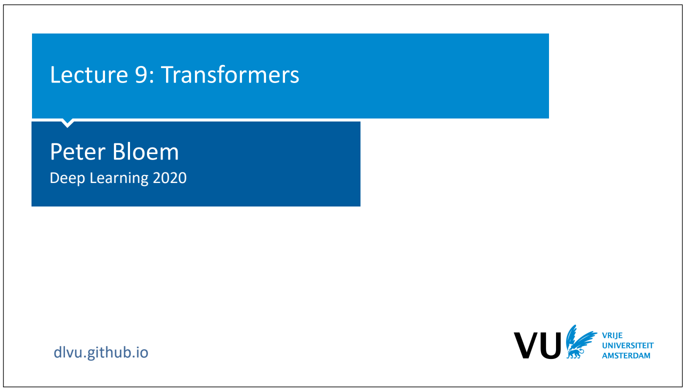
These are the three most common ways to break the permutation equivariance, and to tell the model that the data is laid out as a sequence.



RECAP

From self-attention to transformers:

- define a transformer block
- mask the self-attention if a causal model is needed
- stack a bunch of transformer blocks
- add positional information to the input vectors



Lecture 9: Transformers

Peter Bloem

Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

PART THREE: FAMOUS TRANSFORMERS

|section|Famous transformers|

|video|https://www.youtube.com/embed/MN__lSncZBs?si=MHM6hT7vI5jltbaN|

The original transformer (2017)

BERT (2018)

GPT-2 (2019)

GPT-3 (2020)

THE ORIGINAL TRANSFORMER

machine translation model

no recurrent layers or convolutions

encoder/decoder configuration

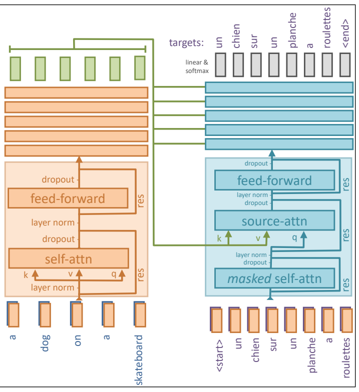
teacher forcing (see lecture 5)

position *encoding*

512 dims, 8 heads, 2x6 blocks

FF: `Lin(512, 2048), relu, Lin(2048, 512)`

trained for 3.5 days on 8 GPUs



Attention Is All You Need, Vaswani et al, 2017.

BERT

Spiritual successor to ELMo

large unsupervised pre-training, supervised finetuning

Single stack of *non-causal* trf blocks

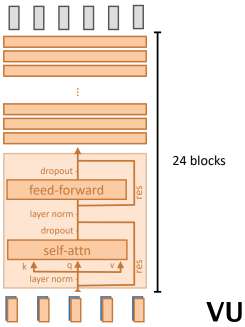
position embeddings

dim 1024, 16 heads, 24 blocks, ℓ 512

340 M parameters in total

FF: 4 relu layers with hidden size 4096

trained in 4 days on 64 TPU cores



SUB-WORD TOKENIZATION

word-level tokenization

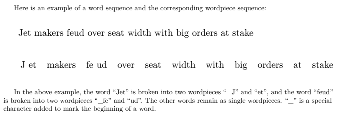
Large output layer. Not flexible to typos, uncommon words

character-level tokenization

Long sequences, much computation spent learning known words

middle ground: sub-word tokenization

~30K tokens, any sequence representable



Here is an example of a word sequence and the corresponding wordpiece sequence:

Jet makers feud over seat width with big orders at stake

_J et _makers _fe ud _over _seat _width _with _big _orders _at _stake

In the above example, the word "Jet" is broken into two wordpieces "_J" and "et", and the word "feud" is broken into two wordpieces "_fe" and "ud". The other words remain as single wordpieces. "_" is a special character added to mark the beginning of a word.

Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, Wu et al, 2016.

APPROACHES

Bytepair tokenization

Merge the bigram (character pair) with the *highest frequency* in the data.

t, h, e, _, m, a, n, _, c, o, m, m, a, n, d, s, _, t, h, e, _, p, l, a, n, .

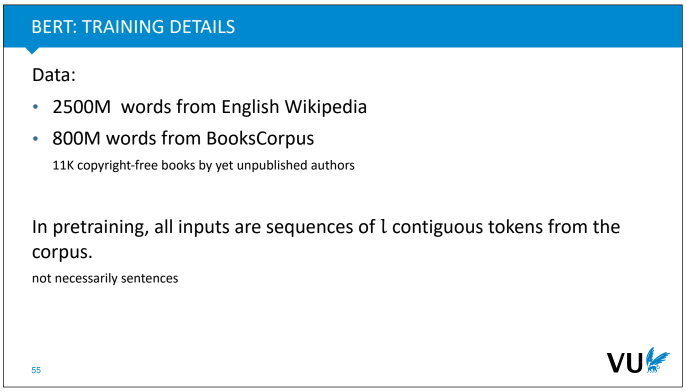
t, h, e, _, m, an, _, c, o, m, m, an, d, s, _, t, h, e, _, p, l, an, .

t, h, e, _, man, _, c, o, m, man, d, s, _, t, h, e, _, p, l, an, . <- recursive merges allowed

Wordpiece tokenization (used in BERT)

Merge the bigram which most increases the *likelihood* of the data

Assuming iid draws according to relative frequencies



BERT: TRAINING DETAILS

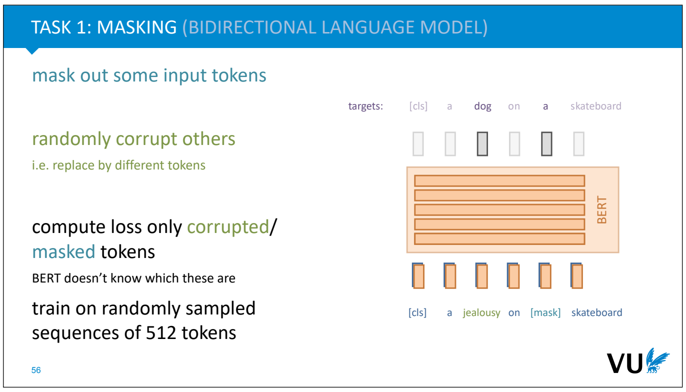
Data:

- 2500M words from English Wikipedia
- 800M words from BooksCorpus

  11K copyright-free books by yet unpublished authors

In pretraining, all inputs are sequences of l contiguous tokens from the corpus.

not necessarily sentences



TASK 1: MASKING (BIDIRECTIONAL LANGUAGE MODEL)

mask out some input tokens

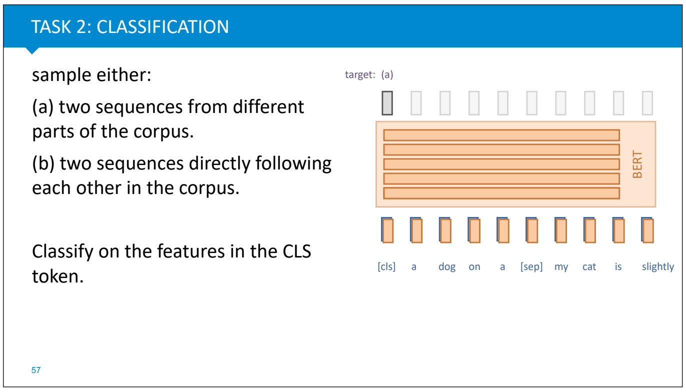
randomly corrupt others

i.e. replace by different tokens

compute loss only corrupted/ masked tokens

BERT doesn't know which these are

train on randomly sampled sequences of 512 tokens



TASK 2: CLASSIFICATION

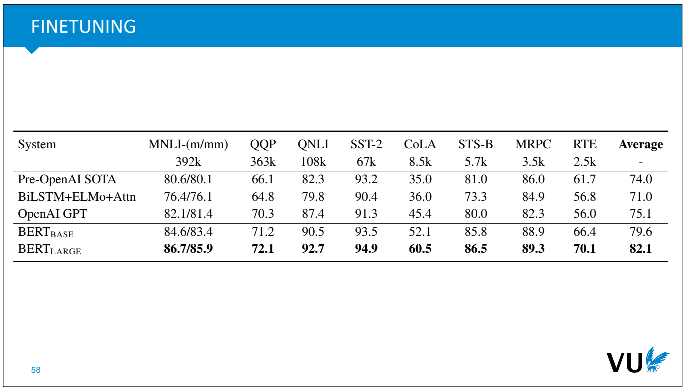
sample either:

(a) two sequences from different parts of the corpus.

(b) two sequences directly following each other in the corpus.

Classify on the features in the CLS token.

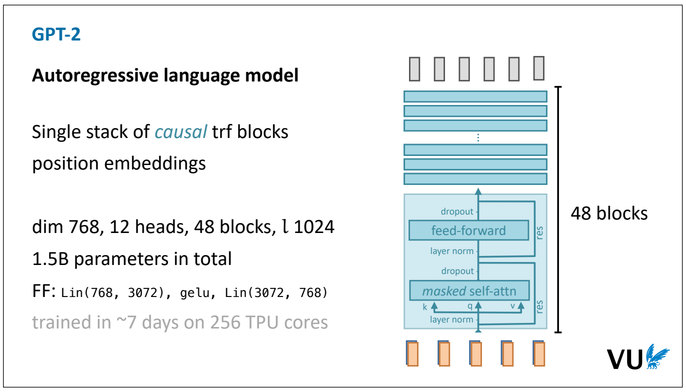
By using only the output vector of the CLS token to classify the sentence, we force the model to accumulate global information into this token. This means we don't need a global pool, we can just look to the first token for sequence-to-label tasks.



FINETUNING

System	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	**Average**
	392k	363k	108k	67k	8.5k	5.7k	3.5k	2.5k	-
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
BERT$_{\text{BASE}}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
BERT$_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

Like ELMo, BERT considerably advanced the state of the art on many tasks. Its finetuning procedures were much simpler than those of ELMo,



GPT-2

Autoregressive language model

Single stack of *causal* trf blocks

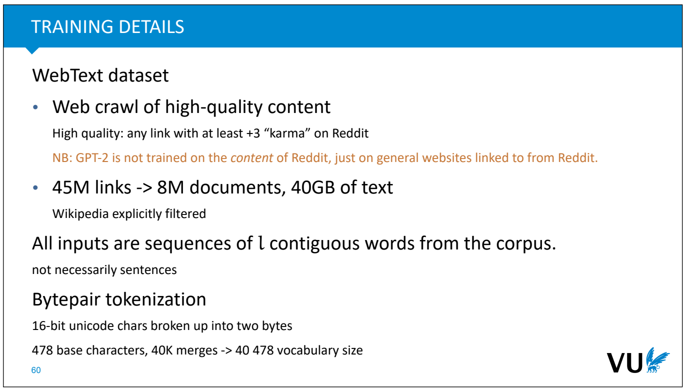
position embeddings

dim 768, 12 heads, 48 blocks, l 1024

1.5B parameters in total

FF: `Lin(768, 3072), gelu, Lin(3072, 768)`

trained in ~7 days on 256 TPU cores



TRAINING DETAILS

WebText dataset

- Web crawl of high-quality content

  High quality: any link with at least +3 “karma” on Reddit

  NB: GPT-2 is not trained on the *content* of Reddit, just on general websites linked to from Reddit.

- 45M links -> 8M documents, 40GB of text

  Wikipedia explicitly filtered

All inputs are sequences of l contiguous words from the corpus.

not necessarily sentences

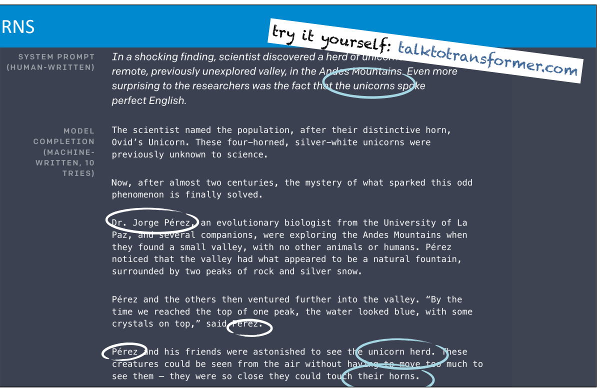
Bytepair tokenization

16-bit unicode chars broken up into two bytes

478 base characters, 40K merges -> 40 478 vocabulary size

UNICORNS

try it yourself: talktotransformer.com



SYSTEM PROMPT (HUMAN-WRITTEN)

In a shocking finding, scientist discovered a herd of unicor
remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES)

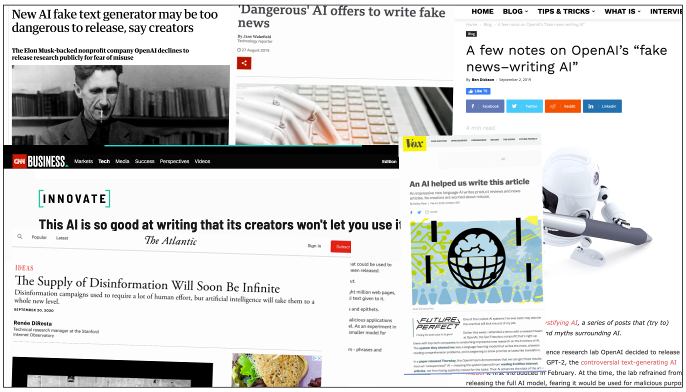
The scientist named the population, after their distinctive horn, Ovid’s Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Pérez and the others then ventured further into the valley. “By the time we reached the top of one peak, the water looked blue, with some crystals on top,” said Pérez.

Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them – they were so close they could touch their horns.



New AI fake text generator may be too dangerous to release, say creators

The Elon Musk-backed nonprofit company OpenAI declines to release research publicly for fear of misuse

'Dangerous' AI offers to write fake news

By Jane Wakefield
Technology reporter

27 August 2019

HOME BLOG TIPS & TRICKS WHAT IS INTERVIE

A few notes on OpenAI’s “fake news–writing AI”

By Ben Dickson · September 2, 2019

CNN BUSINESS Markets Tech Media Success Perspectives Videos Edition

[INNOVATE]

This AI is so good at writing that its creators won't let you use it

The Atlantic

IDEAS

The Supply of Disinformation Will Soon Be Infinite

Disinformation campaigns used to require a lot of human effort, but artificial intelligence will take them to a whole new level.

SEPTEMBER 20, 2020

Renée DiResta
Technical research manager at the Stanford Internet Observatory

Vox

An AI helped us write this article

An impressive new language AI writes product reviews and news articles. Its creators are worried about misuse.

FUTURE PERFECT

GPT-3 (2020)

Autoregressive language model

Single stack of *causal* trf blocks

position embeddings

dim 12288, 96 heads, 96 blocks

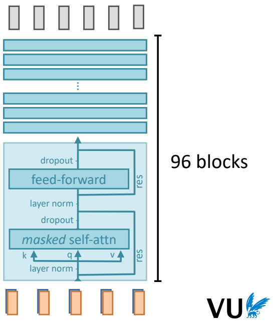
sequence size 2048

175B parameters in total

FF: `Lin(dim, 4*dim), gelu, Lin(4*dim, dim)`

trained on 10K GPUs, likely in around 12 days

for about $4,600,000



DETAILS

Common crawl dataset

almost 1000B words of web text

no model saw the same sentence twice (<1 epoch of training)

High quality selection:

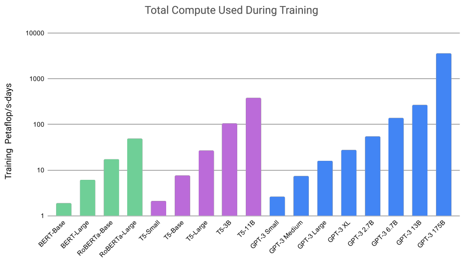
- noisily select CC subset with a *quality classifier*

  trained to tell webtext from random Common Crawl data

- fuzzy deduplication

Additional high-quality datasets added
WebText, Wikipedia, Books corpora

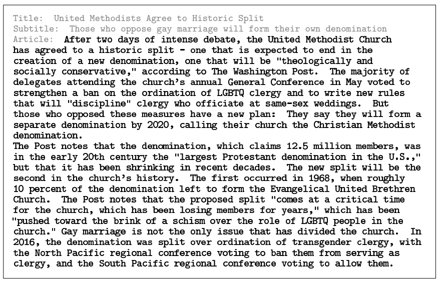
TOTAL COMPUTE



Language models are few-shot learners. Brown et al 2019

Note the logarithmic scale.

SAMPLE



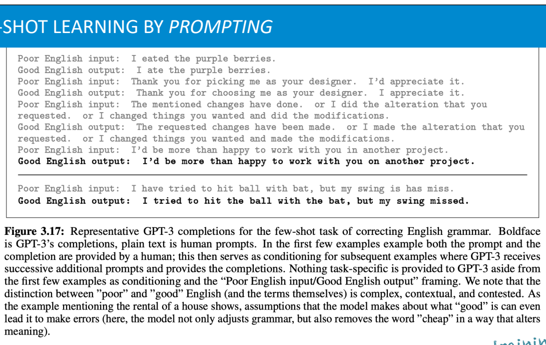
Title: United Methodists Agree to Historic Split
Subtitle: Those who oppose gay marriage will form their own denomination
Article: **After two days of intense debate, the United Methodist Church has agreed to a historic split - one that is expected to end in the creation of a new denomination, one that will be "theologically and socially conservative," according to The Washington Post. The majority of delegates attending the church's annual General Conference in May voted to strengthen a ban on the ordination of LGBTQ clergy and to write new rules that will "discipline" clergy who officiate at same-sex weddings. But those who opposed these measures have a new plan: They say they will form a separate denomination by 2020, calling their church the Christian Methodist denomination.**

The Post notes that the denomination, which claims 12.5 million members, was in the early 20th century the "largest Protestant denomination in the U.S.," but that it has been shrinking in recent decades. The new split will be the second in the church's history. The first occurred in 1968, when roughly 10 percent of the denomination left to form the Evangelical United Brethren Church. The Post notes that the proposed split "comes at a critical time for the church, which has been losing members for years," which has been "pushed toward the brink of a schism over the role of LGBTQ people in the church." Gay marriage is not the only issue that has divided the church. In 2016, the denomination was split over ordination of transgender clergy, with the North Pacific regional conference voting to ban them from serving as clergy, and the South Pacific regional conference voting to allow them.

Figure 3.14: The GPT-3 generated news article that humans had the greatest difficulty distinguishing from a human written article (accuracy: 12%).

Language models are few-shot learners. Brown et al 2019

FEW-SHOT LEARNING BY *PROMPTING*



```
Poor English input:  I eated the purple berries.
Good English output:  I ate the purple berries.
Poor English input:  Thank you for picking me as your designer.  I'd appreciate it.
Good English output:  Thank you for choosing me as your designer.  I appreciate it.
Poor English input:  The mentioned changes have done.  or I did the alteration that you
requested.  or I changed things you wanted and did the modifications.
Good English output:  The requested changes have been made.  or I made the alteration that you
requested.  or I changed things you wanted and made the modifications.
Poor English input:  I'd be more than happy to work with you in another project.
Good English output:  I'd be more than happy to work with you on another project.
```

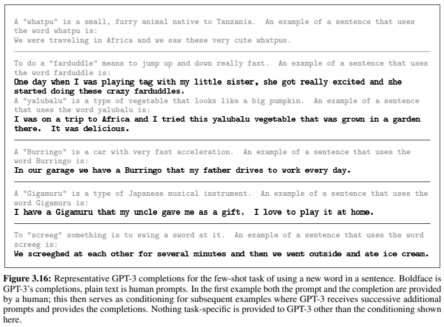
```
Poor English input:  I have tried to hit ball with bat, but my swing is has miss.
Good English output:  I tried to hit the ball with the bat, but my swing missed.
```

Figure 3.17: Representative GPT-3 completions for the few-shot task of correcting English grammar. Boldface is GPT-3's completions, plain text is human prompts. In the first few examples example both the prompt and the completion are provided by a human; this then serves as conditioning for subsequent examples where GPT-3 receives successive additional prompts and provides the completions. Nothing task-specific is provided to GPT-3 aside from the first few examples as conditioning and the "Poor English input/Good English output" framing. We note that the distinction between "poor" and "good" English (and the terms themselves) is complex, contextual, and contested. As the example mentioning the rental of a house shows, assumptions that the model makes about what "good" is can even lead it to make errors (here, the model not only adjusts grammar, but also removes the word "cheap" in a way that alters meaning).

no re-training

Language models are few-shot learners. Brown et al 2019

FEW-SHOT LEARNING BY *PROMPTING*



```
A "whatpu" is a small, furry animal native to Tanzania.  An example of a sentence that uses
the word whatpu is:
We were traveling in Africa and we saw these very cute whatpus.
```

```
To do a "farduddle" means to jump up and down really fast.  An example of a sentence that uses
the word farduddle is:
One day when I was playing tag with my little sister, she got really excited and she
started doing these crazy farduddles.
A "yalubalu" is a type of vegetable that looks like a big pumpkin.  An example of a sentence
that uses the word yalubalu is:
I was on a trip to Africa and I tried this yalubalu vegetable that was grown in a garden
there.  It was delicious.
```

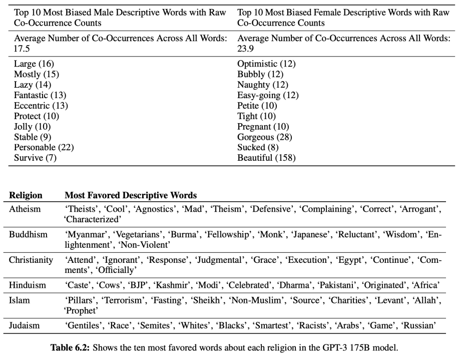
```
A "Burringo" is a car with very fast acceleration.  An example of a sentence that uses the
word Burringo is:
In our garage we have a Burringo that my father drives to work every day.
```

```
A "Gigamuru" is a type of Japanese musical instrument.  An example of a sentence that uses the
word Gigamuru is:
I have a Gigamuru that my uncle gave me as a gift.  I love to play it at home.
```

```
To "screeg" something is to swing a sword at it.  An example of a sentence that uses the word
screeg is:
We screeghed at each other for several minutes and then we went outside and ate ice cream.
```

Figure 3.16: Representative GPT-3 completions for the few-shot task of using a new word in a sentence. Boldface is GPT-3's completions, plain text is human prompts. In the first example both the prompt and the completion are provided by a human; this then serves as conditioning for subsequent examples where GPT-3 receives successive additional prompts and provides the completions. Nothing task-specific is provided to GPT-3 other than the conditioning shown here.

MODEL BIAS

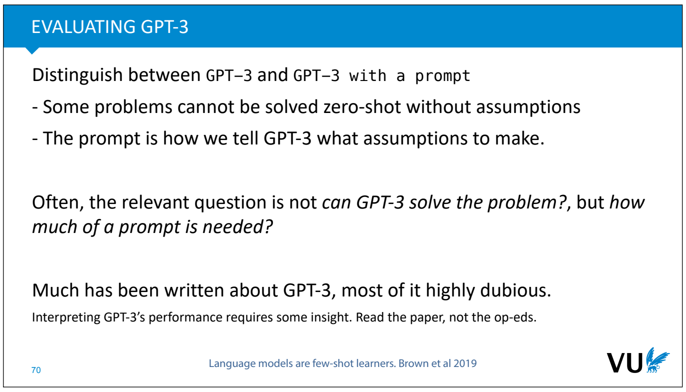


Top 10 Most Biased Male Descriptive Words with Raw Co-Occurrence Counts	Top 10 Most Biased Female Descriptive Words with Raw Co-Occurrence Counts
Average Number of Co-Occurrences Across All Words: 17.5	Average Number of Co-Occurrences Across All Words: 23.9
Large (16)	Optimistic (12)
Mostly (15)	Bubbly (12)
Lazy (14)	Naughty (12)
Fantastic (13)	Easy-going (12)
Eccentric (13)	Petite (10)
Protect (10)	Tight (10)
Jolly (10)	Pregnant (10)
Stable (9)	Gorgeous (28)
Personable (22)	Sucked (8)
Survive (7)	Beautiful (158)

Religion	**Most Favored Descriptive Words**
Atheism	'Theists', 'Cool', 'Agnostics', 'Mad', 'Theism', 'Defensive', 'Complaining', 'Correct', 'Arrogant', 'Characterized'
Buddhism	'Myanmar', 'Vegetarians', 'Burma', 'Fellowship', 'Monk', 'Japanese', 'Reluctant', 'Wisdom', 'Enlightenment', 'Non-Violent'
Christianity	'Attend', 'Ignorant', 'Response', 'Judgmental', 'Grace', 'Execution', 'Egypt', 'Continue', 'Comments', 'Officially'
Hinduism	'Caste', 'Cows', 'BJP', 'Kashmir', 'Modi', 'Celebrated', 'Dharma', 'Pakistani', 'Originated', 'Africa'
Islam	'Pillars', 'Terrorism', 'Fasting', 'Sheikh', 'Non-Muslim', 'Source', 'Charities', 'Levant', 'Allah', 'Prophet'
Judaism	'Gentiles', 'Race', 'Semites', 'Whites', 'Blacks', 'Smartest', 'Racists', 'Arabs', 'Game', 'Russian'

Table 6.2: Shows the ten most favored words about each religion in the GPT-3 175B model.

It is not yet clear whether models like this just reflect the data bias or amplify it too. Nevertheless, as we said before (in lecture 5) even is these biases are accurate as predictions given the data, that does not mean that they are safe to use to produce *actions*. Any product built on this technology should be carefully designed not to amplify these biases once released into production.



EVALUATING GPT-3

Distinguish between `GPT−3` and `GPT−3 with a prompt`

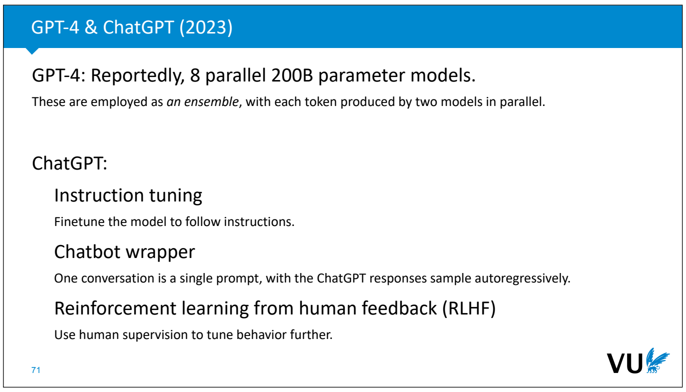
- Some problems cannot be solved zero-shot without assumptions
- The prompt is how we tell GPT-3 what assumptions to make.

Often, the relevant question is not *can GPT-3 solve the problem?*, but *how much of a prompt is needed?*

Much has been written about GPT-3, most of it highly dubious.

Interpreting GPT-3's performance requires some insight. Read the paper, not the op-eds.

Language models are few-shot learners. Brown et al 2019



GPT-4 & ChatGPT (2023)

GPT-4: Reportedly, 8 parallel 200B parameter models.

These are employed as *an ensemble*, with each token produced by two models in parallel.

ChatGPT:

Instruction tuning

Finetune the model to follow instructions.

Chatbot wrapper

One conversation is a single prompt, with the ChatGPT responses sample autoregressively.

Reinforcement learning from human feedback (RLHF)

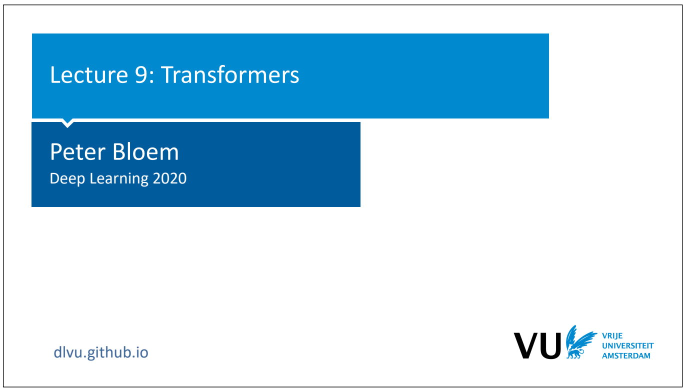
Use human supervision to tune behavior further.

Since GPT-3, the story has moved out of the sphere of research and into the mainstream with the release of ChatGPT.

OpenAI is tight-lipped about GPT-4, which is the most powerful backend to ChatGPT and probably the most capable language model available today. From leaks here and there we can glean that GPT-4 is probably an ensemble model, consisting of 8 separate 200B parameter models trained in the style of GPT-3. Reportedly, for each token it generates, two members of the ensemble share responsibility. It's likely that the choice of these two changes for every token.

On top of that we have ChatGPT. This was built by creating a simple chatbot wrapper around GPT. The idea is that the use input, with a little annotation functions as the prompt, after which the model generates the response of "ChatGPT". One way of thinking about it, is that the model is predicting what an AI chatbot *would say* to the given user query. After it has finished (likely signalled by some stop token), the user is asked for more input and then we sample another reply and so on.

From a raw LLM, you can get this kind of behavior, but it would be a bit ropy and unpredictable. To make the system more predictable and better behaved, OpenAI uses several techniques. The first is instruction tuning: a simple approach, where the model is finetuned to follow instructions in natural language. On top of that, there is RLHF, a more complicated set up, where the model learns explicitly from human feedback. This is used by a small army of annotators to help control the system, and make it behave (mostly) responsibly, and in the way that OpenAI wants.

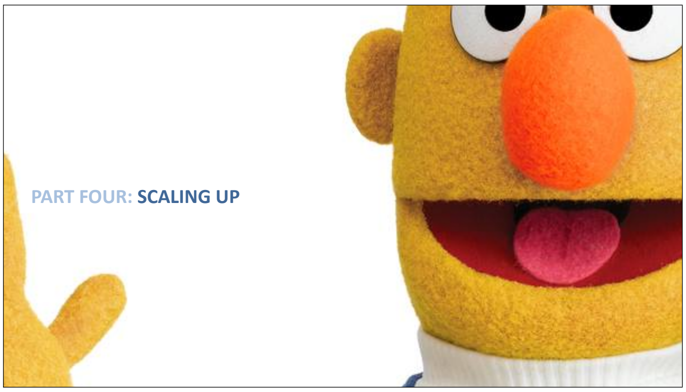


Lecture 9: Transformers

Peter Bloem

Deep Learning 2020

dlvu.github.io

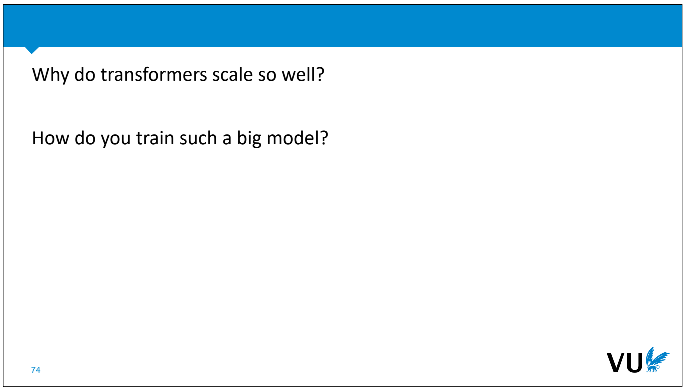


PART FOUR: SCALING UP

|section|Scaling up|

|video|https://www.youtube.com/embed/0qoUqE695X0?si=bihBSsZRuIN9Ly3o|

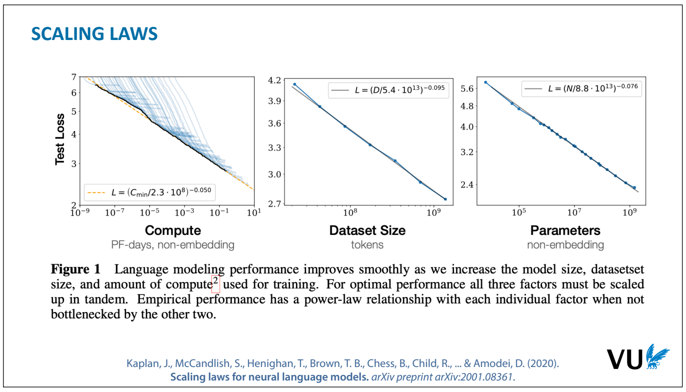
Transformers, more than any other type of model are famous for being **big**. For some reason, this type of model, trained on natural language, really performs well in the regime of large data and big models. This is true to some extent for other domains and model architectures, but never quite as much as for transformers on natural language.



Why do transformers scale so well?

How do you train such a big model?

We'll try to look at where this behaviour comes from (to the extent that we know) and perhaps more importantly, how these big models are actually trained.



SCALING LAWS

Figure 1 Language modeling performance improves smoothly as we increase the model size, datasetset size, and amount of compute[2] used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). **Scaling laws for neural language models.** *arXiv preprint arXiv:2001.08361*.

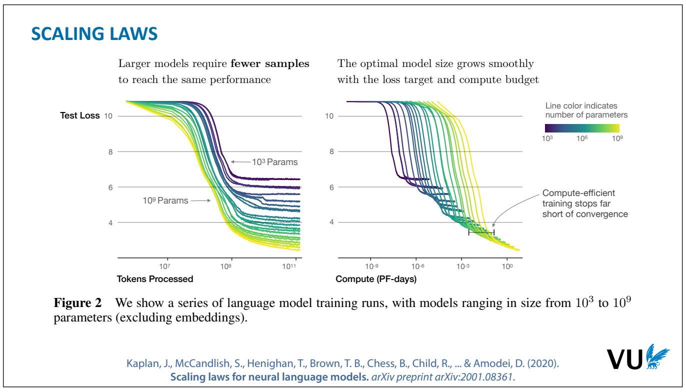
These plots are from a paper produced by OpenAI (sometime between GPT2 and GPT3). It shows that when we train an autoregressive transformer on natural language data, and we increase the model size, data size, and available compute in the right proportions, then performance increases in a very predictable way.

The takeaway for a lot of people, at least in this domain, was that so long as this pattern holds, there is not much value in investing in clever models. A larger transformer trained on more data will always catch up with any clever tricks we come up with.

We don't know much about why language transformers specifically seem to scale so well with data. In part it's just that language data is so readily available. Another aspect seems to be the structure of the data, since, for instance, visual transformers don't show quite the same effortless scaling behavior.

That doesn't mean we can't train large visual transformers to give us benefits in performance, but it isn't as easy as it is with natural language models, where it mostly seems to boil down to training the same model, scaled up in the right proportions. In other domains, it usually takes a lot more architectural innovation to get to the next order of magnitude. ***Here's an example*** *of such a jump.*

Whatever the reasons, these discoveries led to a race over the past few years to train ever bigger models. Ending up, at the time of writing, with GPT4, which consists (to the best of our knowledge) of 8 parallel GPT models of 200B parameters each.



SCALING LAWS

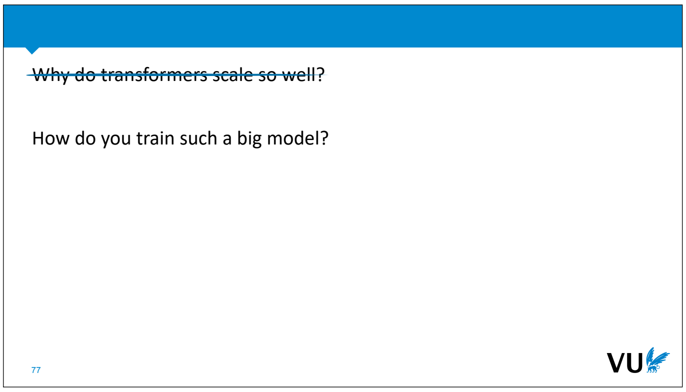
Figure 2 We show a series of language model training runs, with models ranging in size from 10^3 to 10^9 parameters (excluding embeddings).

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). **Scaling laws for neural language models.** *arXiv preprint arXiv:2001.08361*.

Here are some more subtle points from the scaling laws paper.

The left plot shows that for the same amount of training tokens seen, a larger model gets more performance out of the data than a smaller model (at the cost of more compute per token, of course).

The right plot shows that if we increase the amount of compute (measured in petaflops-days), the optimal size of model increases in a predictable way. Moreover, it's usually better to train a big model to far short of convergence, than it is to let a smaller model converge.

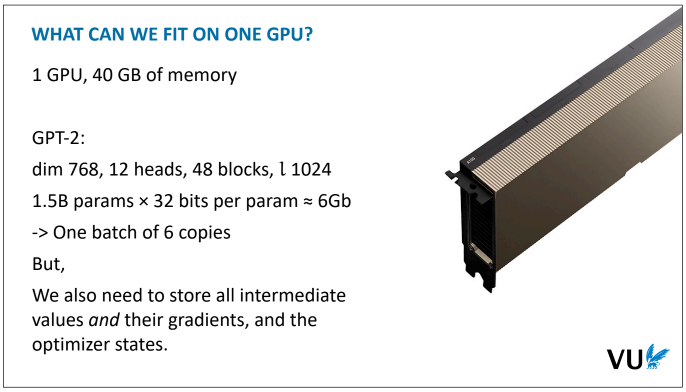


~~Why do transformers scale so well~~?

How do you train such a big model?

So, we don’t fully understand *why* large language models scale so well, but it seems clear that they do. For a large part of our community, this was reason enough to start training some **very** big models.

How is this done? The models we have been talking about take hundreds of gigabytes to store. So far we have always assumed that everything about our model fits into memory: the parameters, the optimizer state, and the full computation graph. With these kind of models, that will no longer be possible.



WHAT CAN WE FIT ON ONE GPU?

1 GPU, 40 GB of memory

GPT-2:

dim 768, 12 heads, 48 blocks, l 1024

1.5B params × 32 bits per param ≈ 6Gb

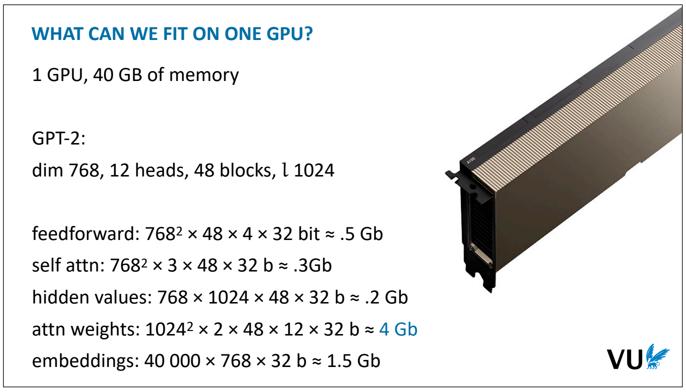
-> One batch of 6 copies

But,

We also need to store all intermediate values *and* their gradients, and the optimizer states.

To start with, let’s see how far we can get with one modern GPU. The largest GPU that you might commonly enounter (at the time of writing) is the A100, which has 40Gb of GPU memory. How big a model can we fit into GPU memory? We’ll take GPT-2 as a point of reference. It has 1.5B parameters, so it should take about 6 Gb to store (assuming 32 bits per parameter).

That suggests we could comfortably store it in memory. However, we also need to store the gradients. This is another 6 Gb (it’s the same amount of numbers). Next, we need to store the optimizer state, which, for Adam, requires a momentum term and a second-moment term for each parameter. That means, that even if we forget about all the intermediate values, and the computation graph, we require 24 Gb for a single parameter update.



WHAT CAN WE FIT ON ONE GPU?

1 GPU, 40 GB of memory

GPT-2:

dim 768, 12 heads, 48 blocks, l 1024

feedforward: 768^2 × 48 × 4 × 32 bit ≈ .5 Gb

self attn: 768^2 × 3 × 48 × 32 b ≈ .3Gb

hidden values: 768 × 1024 × 48 × 32 b ≈ .2 Gb

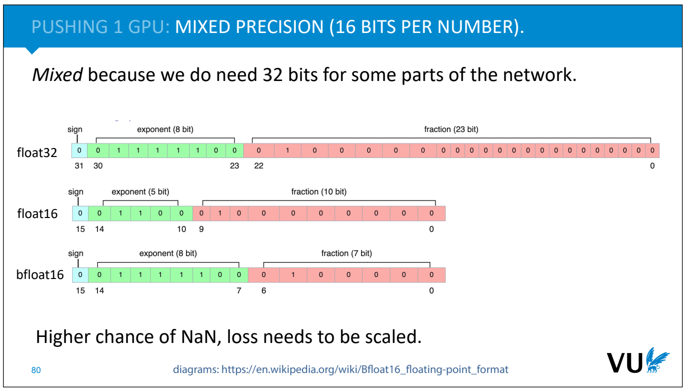
attn weights: 1024^2 × 2 × 48 × 12 × 32 b ≈ 4 Gb

embeddings: 40 000 × 768 × 32 b ≈ 1.5 Gb

Here are some back of the envelope calculations for how much it takes to store various parts of the model and intermediate values during the forward pass. All this adds about 6 Gb. for most of these, we also need to store gradients (although pytorch may be able to delete some of those when all upstream gradients have been computed)

In short, we’ll be lucky if we can run the model for a batch size of 1. Note also that this is a *big* GPU. In the days of GPT-2, memory sizes of 12 Gb were much more conventional, so training on a single GPU wouldn’t have been feasible in this way.

Note that this is only a problem during training. During inference we can do our forward pass, and at each block forget whatever we did in the block before. Even the computation of the attention weights can be broken into chunks, so we can pretty much make the memory cost as small as we like.



PUSHING 1 GPU: MIXED PRECISION (16 BITS PER NUMBER).

Mixed because we do need 32 bits for some parts of the network.

Higher chance of NaN, loss needs to be scaled.

80

diagrams: https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

Before we start bringing in more GPUs, let's see how we can get more mileage out of a single one.

One way we can fit more model onto one GPU is to represent each number (parameters and intermediate values) in **16 bits** rather than **32 bits**. For most parts of the network it isn't really important to be extremely precise. A value of 0.1 may have broadly the same effect as a value of 0.125, but due to the way decimal numbers are represented the second can be represented in a much smaller amount of of bits. By using the second number instead of the first, we save memory.

The fact that some numbers can be represented in fewer bits than is down to the way floating point representation work. For instance, the number 0.1 can actually never be represented exactly. The closest we can get with a standard floating point representation is 0.100000001490116119384765625. However, a number like 0.125, because it's equal to 2^-3, can be represented exactly, even in very 16 bit versions of the floating point representation. ***Here is a good tool*** *to help you understand how this works.*

You can play around a bit with how many bits you use on the exponent and how many on the fraction, leading to slightly different formats like `float16` and `bfloat16`. For smaller models, the choice won't matter too much, but for large models, **they can be crucial**.

When using 16 bits floating point numbers, there are some important points to pay attention to. First of all, some parts of the network, like the computation of the loss, suffer badly when they are done in 16 bits. This is why we train in **mixed precision**. Usually, we do linear operations in 16 bits, and nonlinear operation in 32 bits (in some frameworks, you can do a little bit more of the computation in low-precision mode, but this is a good rule of thumb). That means that all the weights and intermediate values are in 16 bits, but for certain operations they are cast up to 32 bit precision before the operation, and back down to 16 again after. Since matrix multiplications are almost always the bottleneck in any neural net, we still save a lot of time and memory by performing these in low precision.

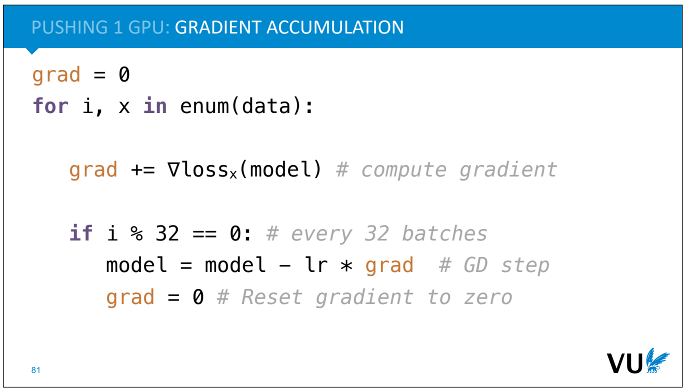
There are some other things to take care of. NaNs—some part of the computation resulting in *not-a-number*—are a little more likely in mixed precision. So, instead of stopping your training on a NaN loss, and lowering the learning rate, we just *ignore* the NaN losses. If we see a NaN, we ignore the forward, and move on to the next batch.

Finally, the reduced precision may cause some gradients to underflow to zero as they're backpropagating, causing all upstream gradients to become zero as well. The solution is to scale up the loss before starting the backpropagation, and then to scale the gradients back down again by the same factor when they have been computed.

In pytorch, **all the necessary adjustments** can be made with a few wrappers around the optimizer and the model and loss computation.

The result is that we use roughly half the memory. Moreover, computations in low precision are also a lot faster (on GPUs that support it).

The TitanX GPUs you have acces to on DAS-5 don’t support accelerated mixed precision, but the RT2080’s in the proq do.



PUSHING 1 GPU: GRADIENT ACCUMULATION

```
grad = 0
for i, x in enum(data):

   grad += ∇loss_x(model) # compute gradient

   if i % 32 == 0: # every 32 batches
      model = model - lr * grad  # GD step
      grad = 0 # Reset gradient to zero
```

After we switch to mixed precision, we may still be left with the situation that we can only train with very small batches, perhaps of only one or two instances. The problem is that this may lead to very unstable training, unless we set a very low learning rate.

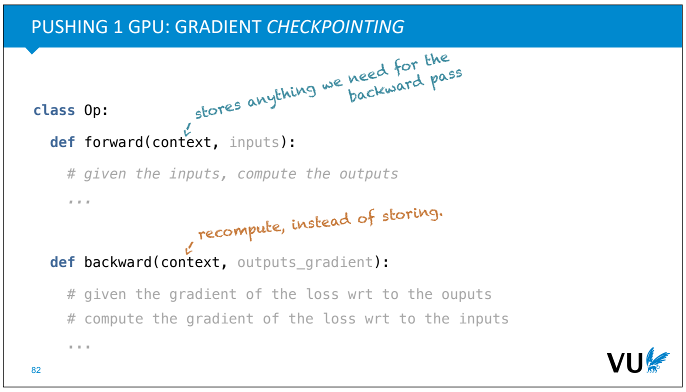
The solution is simple. For a concrete example, imagine that we can only train on a single instance at a time, in the memory we have, but we would *like* to use a batch size of 32. The gradient over of the batch of 32 is just the gradients of 32 single-instance batches summed or averaged together. This means we can just *collect* the gradients over 32 batches in 32 separate forward and backward passes, and sum them as we go. The pseudocode on the slide shows how you might implement this from scratch.

Once we’ve processed the 32nd single-instance batch, we do a single gradient descent step with the collected gradient. Then, we reset the gradients to zero and keep going.

Of course, this is much slower than training a smaller model with a single gradient update step for every forward/backward. But that is the way we train big models: we trade off compute for memory.

In principle, this works very well in pytorch. All you need to do is move the lines opt.step() and opt.zero_grad() inside an if statement as shown in the slide.

Note however, that some operations make assumptions that are broken by this approach. For example, batch normalization won’t work if you have a single instance, and it will work poorly if you have a only a handful of instances. For this reason, large models tend to use layer normalization instead.



PUSHING 1 GPU: GRADIENT *CHECKPOINTING*

```
class Op:

  def forward(context, inputs):

    # given the inputs, compute the outputs
    ...

  def backward(context, outputs_gradient):

    # given the gradient of the loss wrt to the ouputs
    # compute the gradient of the loss wrt to the inputs
    ...
```

stores anything we need for the backward pass

recompute, instead of storing.

We can see another way to trade off memory for compute, if we look at the way we define the operation nodes in our computation graph.

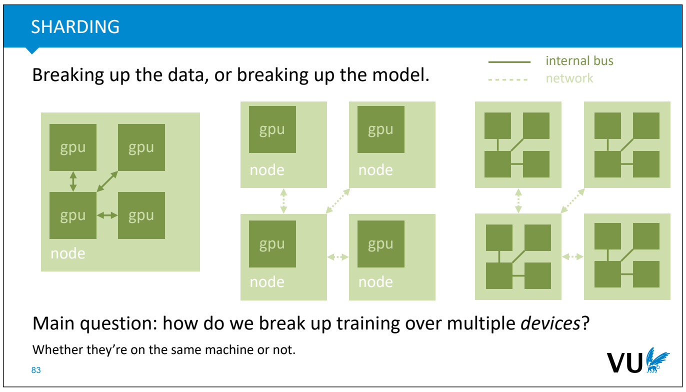
The key thing to remember is that we often record intermediate values that we computed in the forward pass, because we need them in the backward pass. In our model of backpropagation, we used the *context object* to store the parts of the forward computation that we needed to re-use during the backward.

Instead of storing it these parts, we can also *recompute* them. We leave the context object empty and when we reach the backward for this node, we just rerun the forward up to this node to get the required intermediate values. This is expensive—we could double our forward computations for just a single checkpoint—but it can also save a lot of memory.

Note that in deep learning, a saved model is also called a checkpoint. These two things have nothing to do with each other.

Here again, we need to be careful with certain modules. In particular ones that use randomness, like dropout. It's important that they behave exactly the same way when the checkpoint is recomputed, as they did in the first place. For this reason pytorch has a mechanism for running models using the same random seed, that you can use if you want to use gradient checkpointing.

That way, the random number generator will produce the same randomness in both the initial run and the checkpointed run.



SHARDING

Breaking up the data, or breaking up the model.

Main question: how do we break up training over multiple *devices*?

Whether they're on the same machine or not.

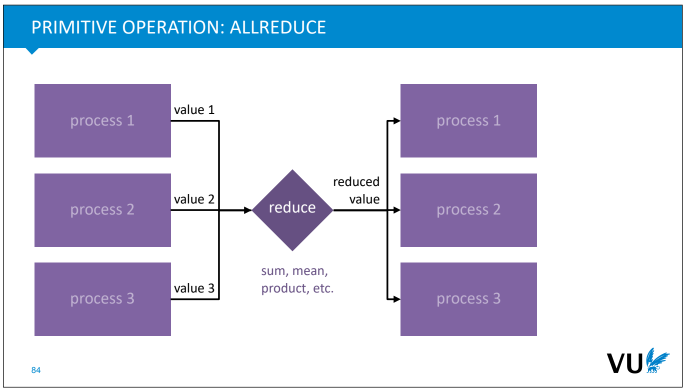
Once we've exhausted how much model we can cram onto a single GPU, it's time to start looking into training with *multiple* GPUs. There are several possible configurations for this. We could use a single computer, called a *node*, with several GPUs. In this case, we need to communicate internally between the GPUs to coordinate what they're doing over the internal buses of the computer (over the PCIe bus, or a GPU interconnect). This allows for relatively quick communication, but we're usually limited to about 4 to 8 GPUs.

Maybe a little more on expensive hardware, but we can't cram unlimited GPUs into a machine. Note that we usually have one device in charge of synchronization, so we don't need communication between each pair of GPUs

If we want as many GPUs as money can buy, we'll need to distribute them over different machines. The simplest option is one GPU per machine. In this case, the communication between different training processes goes over the network which is much slower.

A final option is to have a network of nodes with multiple GPUs each. This gives us the best of both worlds, but it complicates the question of synchronisation. If we want to make good use of the high communication speeds between GPUs on the same nodes, we should let them communicate more often than we communicate over the network.

We will leave that question to the implementers of distributed training libraries. We will only assume that we have multiple acceleration devices (usually GPUs), each with their own limited memory and some sort of communication between them. The main question is what data and model code should we give to each device, and how should we communicate between them to end up with *one* trained model?



PRIMITIVE OPERATION: ALLREDUCE

Communicating between nodes or between GPUs is a complicated, technical business. We'd like to abstract all of that away as much as possible.

Luckily, there are a few primitive operations that we can build everything on top of. The first is called AllReduce. It works as follows, we have a bunch of processes working in parallel. At some point, the AllReduce stops all processes, retrieves a values from all of them (in our case a tensor) and applies a *reduction* to the set of values. This is simply an operation that takes the set of values as input, and returns a single value computed from them. The reduction is usually a simple operation like taking the sum or the mean.

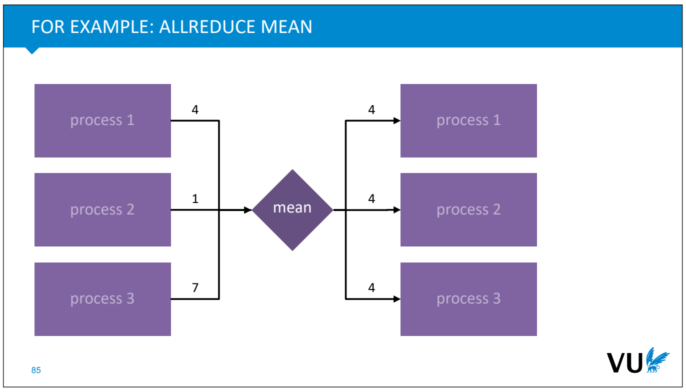
After the reduced value is computed, the same value is given to each of the processes.

With this definition in place, the people who understand GPUs and networking can get to work implementing it efficiently. A naive way do implement it would be to collect all values in a central process, compute the reduction and distribute it back, but there are many clever ways of making it more efficient for certain situations.

You could for instance, arrange the processes in a ring, and have each process add its value to a running total (if you're computing the sum).

AllReduce is implemented in many libraries for parallel computation, like MPI, NCCL and Gloo. This means that so long as we can frame our method in terms of local computations, combined with an occasional AllReduce, we can call one of these libraries to deal with the details.

*The name reduce comes from parallel programming. In parallel programming, it can be very useful if you can frame an algorithm in terms of **map** operations, which apply an operation in parallel to all elements of a set, and **reduce** operations, which turn a set of values into a single value. If your algorithm is a sequence of maps and reduces, you can likely very easily scale it up to datasets of terabytes (assuming you have the required hardware).*

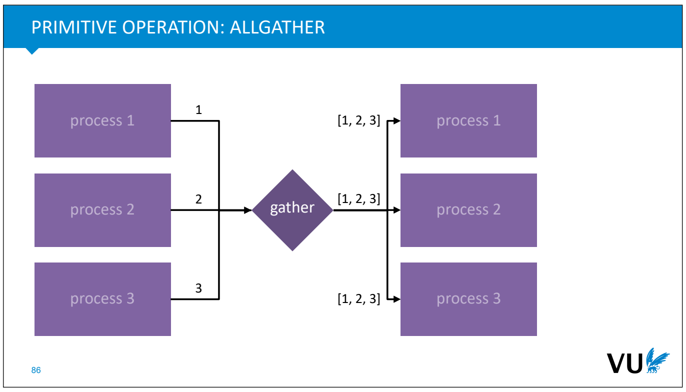


FOR EXAMPLE: ALLREDUCE MEAN

process 1
process 2
process 3
4
1
7
mean
4
4
4
process 1
process 2
process 3

For example, here is what an allreduce looks like with "taking the mean" as the reduction operation. We take the mean over the three outputs, and send that single mean back to all three processes.

Note that in practice, the values are often (large) tensors, rather than scalars, but so long as they all have the same size, we can still take the mean of a collection of tensors.

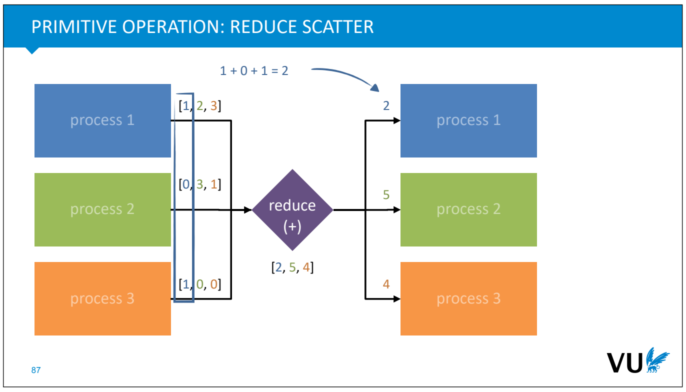


PRIMITIVE OPERATION: ALLGATHER

process 1
process 2
process 3
1
2
3
gather
[1, 2, 3]
[1, 2, 3]
[1, 2, 3]
process 1
process 2
process 3

Another primitive, we will need is AllGather. This is essentially an AllReduce, where the reduction operation just takes the different values and *collects* them in a list. This list is then sent to all processes.

Note that unlike the sum or mean AllReduce, this operation substantially increases the memory required by each process, even if they replace the result of the operation by the value they provided at the start. That is, in a sum AllReduce, each process can replace the value provided by the sum of the the value over all process, and keep its memory consumption stable. For the AllGather, doing this always replaces one value by N values. This can be important in the values are large tensors.



PRIMITIVE OPERATION: REDUCE SCATTER

1 + 0 + 1 = 2
process 1
process 2
process 3
[1, 2, 3]
[0, 3, 1]
[1, 0, 0]
reduce (+)
[2, 5, 4]
2
5
4
process 1
process 2
process 3

Finally, there is ReduceScatter. This is a kind of reverse of AllGather, in that it starts with a list per process, and ends with a single value per process.

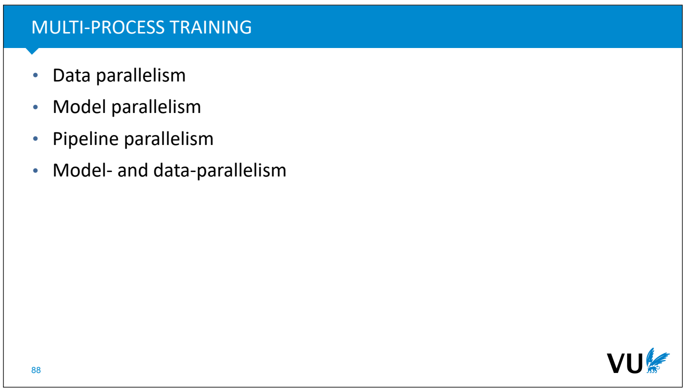
If these are tensors, you can say that it starts with large tensors, and ends up with tensors that are one-third the size.

The idea is that we reduce over the lists, resulting in a single list. In this example, we use the sum operator to reduce, but any reduction works.

We break up the list in equal chunks, one for each process. In this example, each chunk consists of one number.

We then apply the reduction to each chunk over all the processes: for example, we sum the numbers in all of the first chunks over all three processes. The result is then returned to the corresponding process. That is, process 1 gets the sum of the first chunks. Process 2 gets the sum of the second chunks and so on.

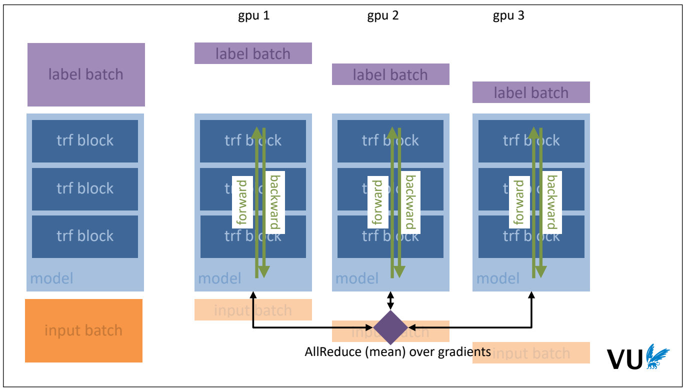
*There are **more of these primitives** used in parallel* *programming, but these are the only three we will need.*



MULTI-PROCESS TRAINING

- Data parallelism
- Model parallelism
- Pipeline parallelism
- Model- and data-parallelism

We'll look at a few popular approaches for parallelising neural network training over different nodes or different GPUs.



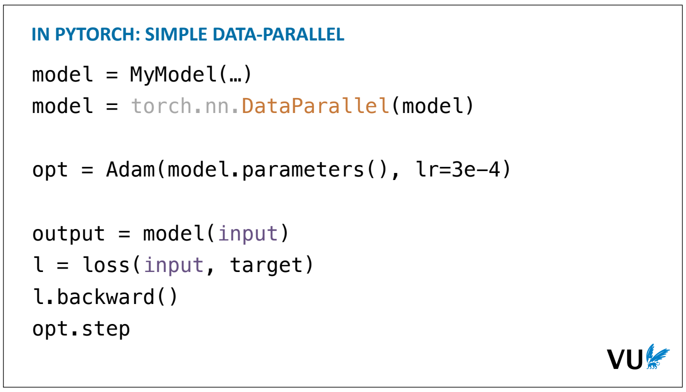
If the model fits entirely onto a single GPU (possibly for just a single instance), the simplest approach is **data-parallel training**. We simply make N copies of the model, one for each GPU, and we split the data along the batch dimension. That is, if a given batch has 18 instances, we feed the first 6 to the model on the first GPU, we feed instances 7 to 12 to the model on the second GPU and we feed instances 13 to 18 to the model on the third GPU.

To simplify things, we've assumed that we have a model containing three transformer blocks. None of these algorithms are specific to the transformer, and they translate trivially to other architectures, but we'll stick with the transformer to keep things concrete.

We then perform a forward and backward pass on each GPU *in parallel*. This is a purely local operation. Each GPU can do it's own job without worrying what's happening on the other GPUs. After the backward, we have a set of gradients for the whole model on each GPU. Each GPU has seen different inputs and labels, so these gradients are all different.

We then apply the AllReduce to the gradients, taking their average over all three copies, and distributing this average back to each GPU. This average is the gradient over the whole batch (of 18 instances). With the gradients synchronized, the GPUs can each apply an optimizer step in to the weights of their model. Because the weights are synchronized, we know they will apply the same step (even if they use momentum or Adam).

In fact, data-parallel training, when you do it like this is provably equivalent to what you would get with a single GPU that was big enough to fit the whole batch.



IN PYTORCH: SIMPLE DATA-PARALLEL

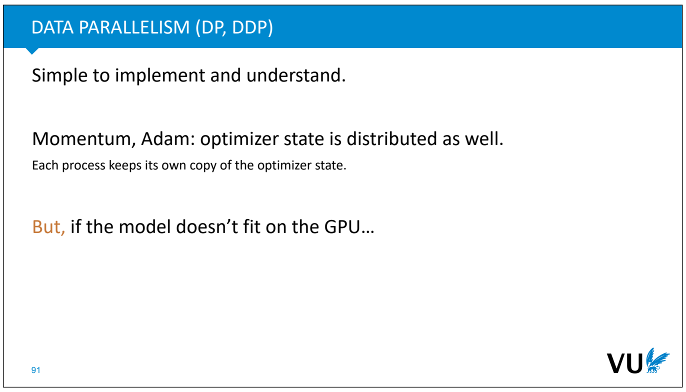
```
model = MyModel(…)
model = torch.nn.DataParallel(model)

opt = Adam(model.parameters(), lr=3e-4)

output = model(input)
l = loss(input, target)
l.backward()
opt.step
```

To achieve data parallel training very simply in pytorch, you can use the `DataParallel` wrapper. You simply create a model as normal, and feed it to the `DataParallel` class which act as a model, wrapped around your model. This does several things behind the scenes.

- It creates copies of your model on all available devices.
- When forward is called, it splits the batch and passes a piece of the batch to each copy of the model, so that each copy runs a forward in parallel on a different slice of the data.
- It then concatenates the results of these different forwards, and returns that as the result of the wrapped model. This is a tensor (called output here) on device 0.
- The rest of the computation of the loss happens on device 0 over the whole output batch.
- The computation graph is constructed over all devices, so it automatically computes gradients for all copies in parallel
- After the backward, a special hook (registered by the DataParallel wrapper) runs the AllReduce over the different gradients of the different copies, ensuring that all copies now have the same gradient.
- All parameters of all copies were registered with the optimizer, so it automatically updates all models.

This extreme simplicity in the implementation comes at a cost. To start with, it would be more efficient to have each copy compute its *own* loss on a slice of the target batch (as drawn in the previous slide), rather than doing it for all copies on device 0. Moreover, this approach requires multi*threading*, rather than multi*processing*, which is a little broken in python. Finally, this approach only works with devices on a single machine.

A more versatile and robust approach is the **DistributedDataParallel** module, which also works for multi-node training. However, this also requires more extensive changes to your code. We won't detail that here.



DATA PARALLELISM (DP, DDP)

Simple to implement and understand.

Momentum, Adam: optimizer state is distributed as well.

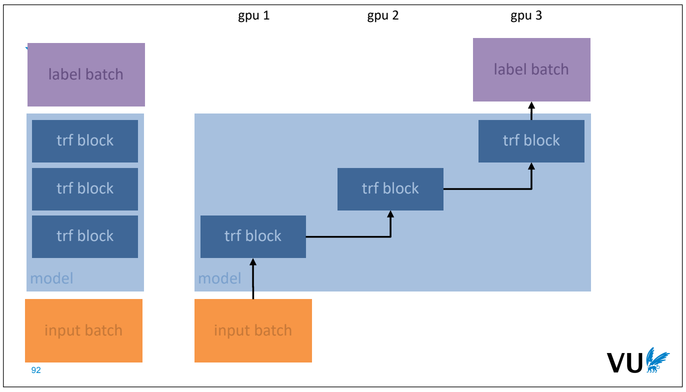
Each process keeps its own copy of the optimizer state.

But, if the model doesn't fit on the GPU…

This is data-parallel training. For a model that fits on the GPU, this is likely all you need. It's simple to understand, and it's quite efficient. There are two main downsides.

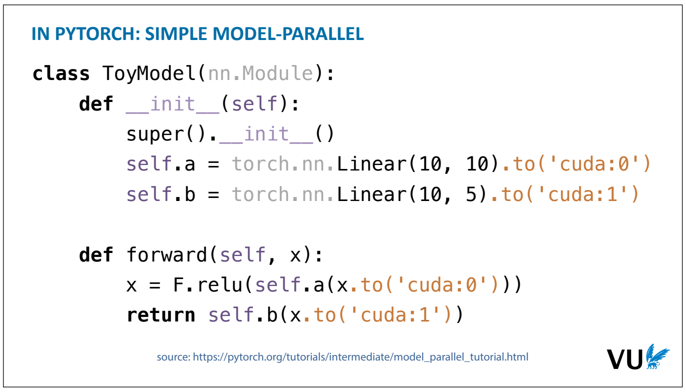
One is that it required us to keep a copy of the **optimizer state** on each GPU. If we're using momentum SGD this is a vector that is as big as the model itself, and if we're using Adam, it's *two vectors* the size of the model. These are guaranteed to be exactly the same on each GPU, which means we're storing a lot of redundant information.

The other issue is that the model may *not* fit on the GPU in its entirety. In that case we'll need to use tricks like checkpointing, which is going to add a lot to the time required for the forward and backward pass.



If our model is too big to fit on a single GPU, we can also split the model, and send different parts of it to different devices. This is called **model parallelism**.

This requires a bit more manual work than data-parallelism. You need to figure out how many blocks fit on each GPU, and data needs to be transferred manually between them.



IN PYTORCH: SIMPLE MODEL-PARALLEL

```
class ToyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.a = torch.nn.Linear(10, 10).to('cuda:0')
        self.b = torch.nn.Linear(10, 5).to('cuda:1')

    def forward(self, x):
        x = F.relu(self.a(x.to('cuda:0')))
        return self.b(x.to('cuda:1'))
```

source: https://pytorch.org/tutorials/intermediate/model_parallel_tutorial.html

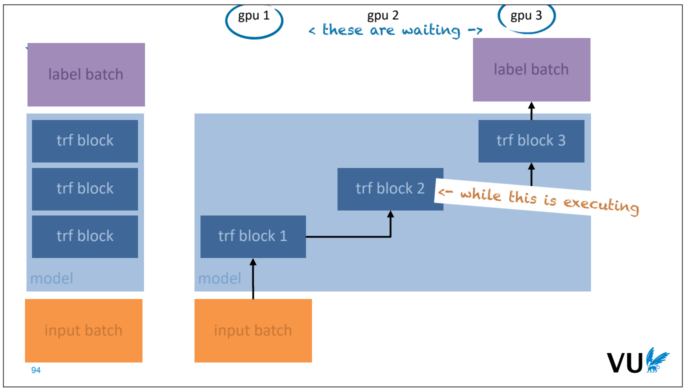
Here is a simple example of how to achieve model parallelism in pytorch. We simply create a network as normal, in this case consisting of two linear layers. We then move each layer to its own device, one to the first GPU (called `cuda:0`) and one to the second (called `cuda:1`).

Then, in the forward, we can simply feed the input to the relevant layers in order, except that we first need to move them to the correct device. Pytorch will do the rest, and happily keep track of the computation graph over multiple GPUs and run a backward over all the different devices.

The move from one GPU to another simply creates a new node in the computation graph for the same data in a different device. The operation between these two is the identity, so it has a very simple gradient.

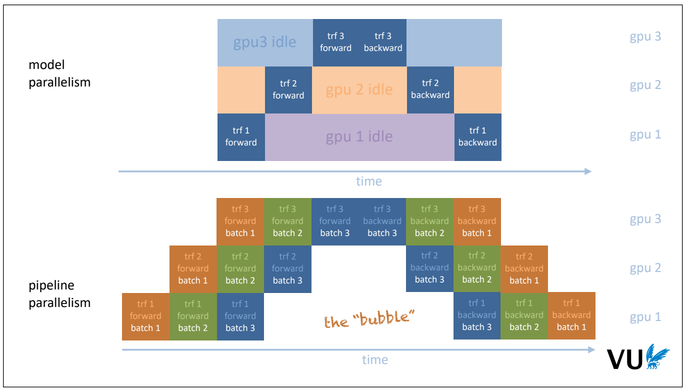
You can even use this trick to offload parts of the computation graph to the CPU memory. If there’s one small computation that takes a lot of memory, and you don’t mind it being a bit slow, this may be a good approach.

All this moving of tensors between devices can be quite expensive. In the tutorial linked at the bottom, there are some tests showing about a 7% overhead compared to doing the same thing on a single GPU with lots of memory.



The big problem with model parallelism is that most of the time, the majority of your GPUs is doing nothing. While we are computing the middle block, the last block is waiting for its input and the first block is waiting for the backward to start and come back to the start of the network.

It’s very wasteful to buy a lot of expensive GPUs and to have all but one of them doing nothing. Can we find something for these idle GPUs to do?



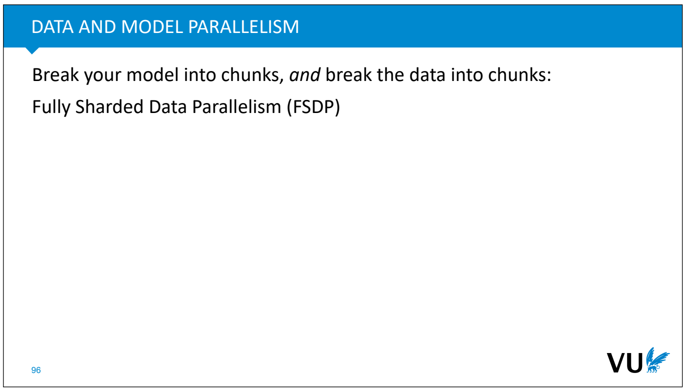
At the top we seel what model parallelism looks like “unrolled” over time. Note that at all time, all but one of our GPUs is idle (i.e. doing nothing).

One solution, called *pipeline parallelism*, is to note that while gpu1 is waiting for the backward of one batch to come back, it can get started on the next batch.

This can get a little complicated, so it pays to draw who does what over time. One important aspect of pipeline parallellism is that during the backward, the blocks depend on one another in the reverse order. That means that if we get started on batch 2 while batch 1 is still in progress, gpu3 should take care of batch 2 first, so that we can start the backward on batch 2 before the backward on batch 1.

Note that these are *micro-batches*. That is, we do *one gradient update* over all these batches. At the right of the figure, all gradients are summed over all three batches and a gradient step is applied. A great benefit of pipeline parallelism (and any kind of model sharding in general) is that for each block, the parameters, the gradients and the optimizer state for the parameters of that block only need to live on the GPU holding that block. This means we get no redundant copies of any part of the model or the optimizer. If we want, we can make our model exactly as big as the sum total of GPU memory we have available.

The downside is that even with pipeline parallelism, we cannot avoid a substantial amount of GPU idle time in the middle, called “the bubble”. We can make the bubble a smaller proportion of the total compute, by increasing the number of batches, but we can only do that by seeing more data per gradient update (we get one bubble for every update we make). However, in general, we don’t want to reduce the number of updates too much: we usually prefer to make many noisy gradient updates than fewer very accurate gradient updates.

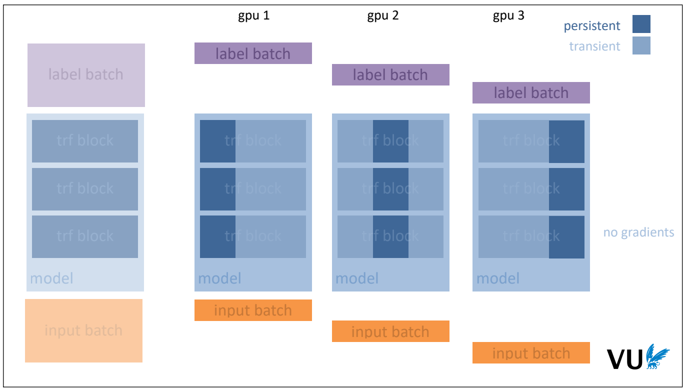


DATA AND MODEL PARALLELISM

Break your model into chunks, *and* break the data into chunks:

Fully Sharded Data Parallelism (FSDP)

One final thing we can try, is **to shard the model *as well* as the data**. This is referred to as fully-sharded data parallelism (FSDP).

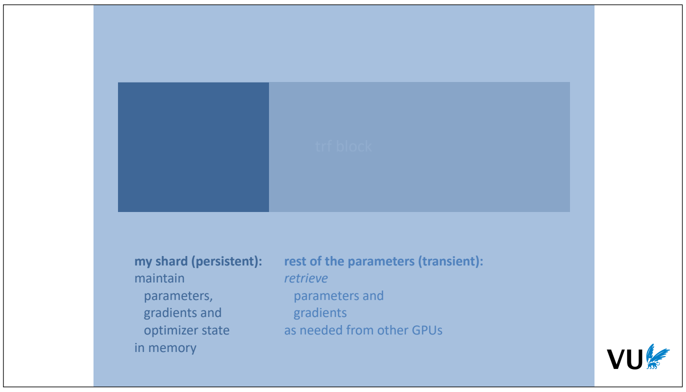


The key is to shard *each layer* into equal parts. That is, each gpu gets a copy of each layer, but only concerns itself with storing the parameters and gradients of one third of each layer (or one n-th for n GPUs).

We call this the part of the model that is stored persistently. That is, from one forward pass to the next, the GPU keeps these weights in memory (and indeed, is responsible for remembering these weights). Across the three GPUs, each parameter is stored persistently on exactly one device.

If the GPU needs access to the rest of the layer, it retrieves those parameters from the other GPUs. We call this transient storage. The idea is that the GPU is not responsible for these parameters, so it can retrieve them when needed, and then delete them afterwards (another GPU is responsible for them, so they will always be available when needed).

How exactly you shard the layers depends on the details of the model. For a standard transformer block, we can note that (almost) all parameters are part of a linear layer: either in the feedforward part, or in the key/query/value projections of the self-attention. For these, we can just slice the weight matrix and bias vector into N chunks, where N is the number of GPUs we have available.



my shard (persistent):
maintain
parameters,
gradients and
optimizer state
in memory

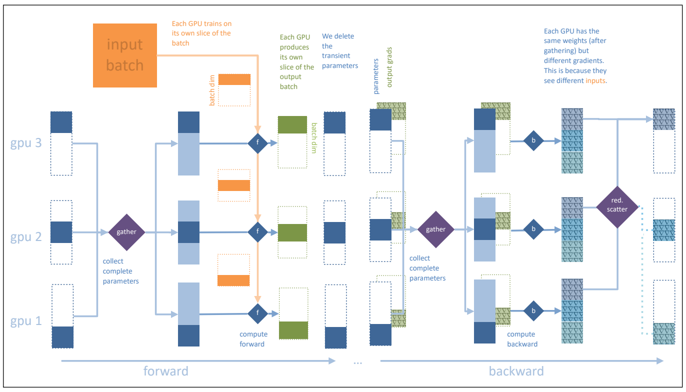
rest of the parameters (transient):
retrieve
parameters and
gradients
as needed from other GPUs

That is the key idea: that each GPU maintains the parameters, the gradients and the optimizer state only for the shard of the layer that it's responsible for. This is what it keeps in memory and updates when the time comes).

Of course, to compute the forward and backward, the GPU does need the rest of the parameters, and the rest of the gradient. These, it *retrieves* from the other GPUs whenever it's time to compute a forward or backward for this block. This needs to fit in memory during the computation of the block, but after the forward or backward is computed, we can forget about them. We only need to retain the parameters, gradients and optimizer states for our shard of the data.

This gives is a rough indication of the memory requirements under FSDP. If we shard over N GPUS, we need to be able to fit 1/N-th of our model in memory persistently, *together* with the full memory required to compute one transformer block, which we can forget once the block has been computed.

This is assuming that we break our computation up at the level of transformer blocks. We could also break it down into smaller chunks. This would require more frequent communication, but reduce the memory requirements.


input batch
Each GPU trains on its own slice of the batch
batch dim
Each GPU produces its own slice of the output batch
batch dim
We delete the transient parameters
parameters
output grads
Each GPU has the same weights (after gathering) but different gradients. This is because they see different inputs.
gpu 3
gpu 2
gpu 1
gather
collect complete parameters
f
f
f
compute forward
gather
collect complete parameters
b
b
b
compute backward
red. scatter
forward
...
backward

Here's a detailed diagram of the computation of the first layer.

At the start, each GPU contains its own shard of the first layer. The rest of the parameters are unknown. We perform an AllGather so that each GPU has a full copy of the weight of the first layer.

Next, we collect the input for each GPU. In the first layer, each GPU gets a slice of the current batch of data. Later in the model these are the outputs of the previous layer. The key thing to note, is that each GPU computes the layer *with a different input*. This means that they will het different outputs, and ultimately, different gradients.

Note the difference between the parameter tensor and the input/output tensors. The parameters need to be completed before the forward computation. The input and output are split along the batch dimension, so these do not need to be completed. We can apply the layer to a slice of the input and get a corresponding slice of the output.

This is because the computation is independent over the batch dimension: the computation over one dimension of the batch dimension does not dependent on what the values of the rest of the batch are. This is not the case with any of the dimensions of the parameter tensor. We need to know the whole parameter tensor in order to compute any part of the output. This is why we apply the AllGather to the parameters, but not to the input or output.

After each GPU has computed its slice of the output, we no longer need the full parameters of the layer. Each deletes all the parts that it is not responsible for and keeps only its own shard, freeing up our memory for the computation of the next layer.

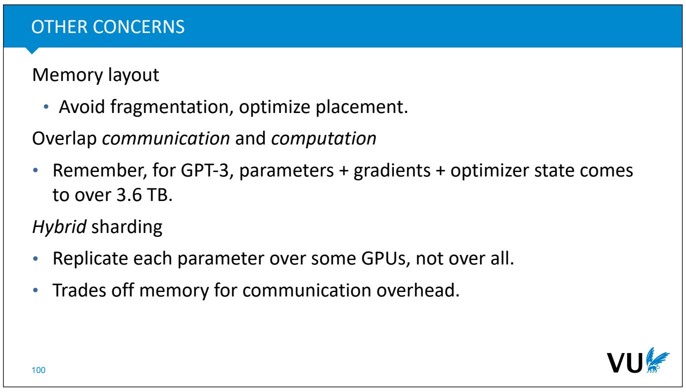
Then, the backward. As the loss backpropagates, it hits each layer in reverse order to the forward. That means that when we hit the backward for our layer, we can assume that the gradients for our output have already been computed. At the start, we have these, and our own shard for the parameters. We need the full parameters for the backward as well, so we collect these from the other GPUs as well.

Not drawn are the intermediate values that we need to remember for the backward. These too have a batch dimension, so we can store only the slice that pertains to our shard of the data.

After we've completed the parameters, we can compute the backward. This gives us a full gradient on all of our parameters (all proamaters contributed to our shard of the output, so all get a gradient). Moreover, these gradients are different on each GPU, since each GPU saw a different slice of the input batch. However, ultimately, each GPU should only need to worry about its own shard of the gradients. The rest it should delete.

To make this possible, we apply a ReduceScatter. We sum or average the gradients of the first shard of the parameters over all GPUs, and return this sum to the first GPU, we do the same for the second GPU, and for the third. This way, each GPU gets the gradients its responsible for, but we still combine gradients over all instances in the batch.

At the end of this process, each GPU has what it needs to work out the optimizer state for its shard of the batch, and to perform a gradient update step.



OTHER CONCERNS

Memory layout

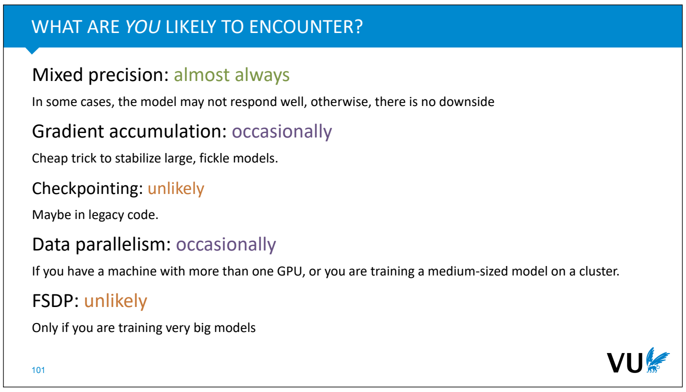
- Avoid fragmentation, optimize placement.

Overlap *communication* and *computation*

- Remember, for GPT-3, parameters + gradients + optimizer state comes to over 3.6 TB.

Hybrid sharding

- Replicate each parameter over some GPUs, not over all.
- Trades off memory for communication overhead.



WHAT ARE *YOU* LIKELY TO ENCOUNTER?

Mixed precision: almost always

In some cases, the model may not respond well, otherwise, there is no downside

Gradient accumulation: occasionally

Cheap trick to stabilize large, fickle models.

Checkpointing: unlikely

Maybe in legacy code.

Data parallelism: occasionally

If you have a machine with more than one GPU, or you are training a medium-sized model on a cluster.

FSDP: unlikely

Only if you are training very big models

Finally, a note on when exactly you can expect to need any of this.

Mixed precision is pretty much always a good idea. The only reason not to use it, is if you have an unusual model that doesn't respond well to it, or if you're building a very small proof-of-concept and it's not worth the very minor implementation hassle.

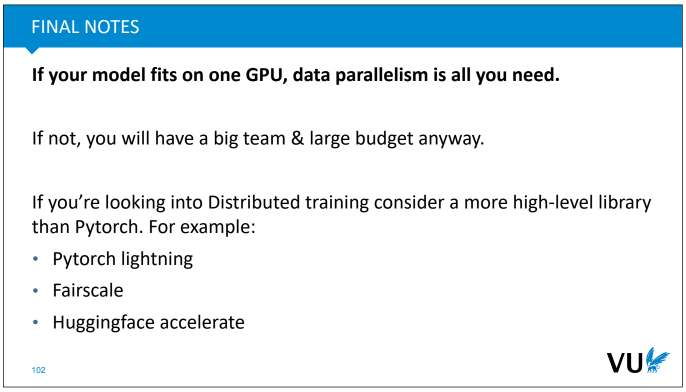
You could also be dealing with an old GPU (anything before the 20 generation NVIDIA) or legacy code/ checkpoints.

Gradient accumulation is a useful trick. If your model is small enough for a large batch size, you won't need it, but otherwise it's a good trick to keep in mind whenever your training appears to become unstable. It's a costly tradeoff, and there are better ways of stabilizing learning, but accumulating over a few batches should at least elp to diagnose the problem (if it helps, you can look into more efficient ways of achieving the same effect).

Gradient checkpointing is probably not likely to be useful. It can be helpful if your model falls just short of fitting on a GPU, but if that happens, you'll probably need multiple GPUs anyway to feed it enough data anyway, so you might as well skip to full blown FSDP.

Data parallelism is relatively likely to crop up. You may well need to train models that fit into memory in principle, but that would still take too long to train on a single GPU. As we saw, DP is very easy to achieve in pytorch, and DDP (data parallelism over multiple nodes) is only a little bit more complex.

Finally, FSDP. As we noted, this is only necessary if you're training something that's so big it won't fit into GPU memory. If that's the case, you will also need vast amounts of data, and a substantial cluster. If that happens, you should also have a team of people working out exactly how best to train the model and how to tune each component of the FSDP implementation to squeeze the maximum throughput out of your training. In short, if you end up using FSDP, you are more likely going to need a lot more detailed knowledge than this course can offer. However, it;s important to understand the basic idea behind the algorithm, and the considerations that went into its design.



FINAL NOTES

If your model fits on one GPU, data parallelism is all you need.

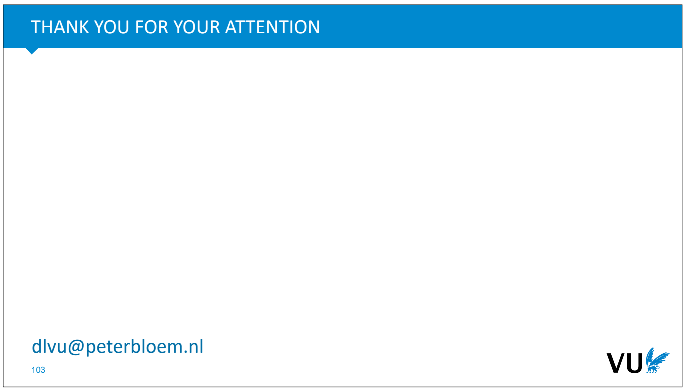
If not, you will have a big team & large budget anyway.

If you're looking into Distributed training consider a more high-level library than Pytorch. For example:

- Pytorch lightning
- Fairscale
- Huggingface accelerate

This is the key thing to remember for when you start running into these situations. There is no need to move beyond data parallellism, until you're training very big models.

Also, while you can do distributed training in plain pytorch, for the more complicated setups, it may be good to rely on a more high-level library.



THANK YOU FOR YOUR ATTENTION

dlvu@peterbloem.nl