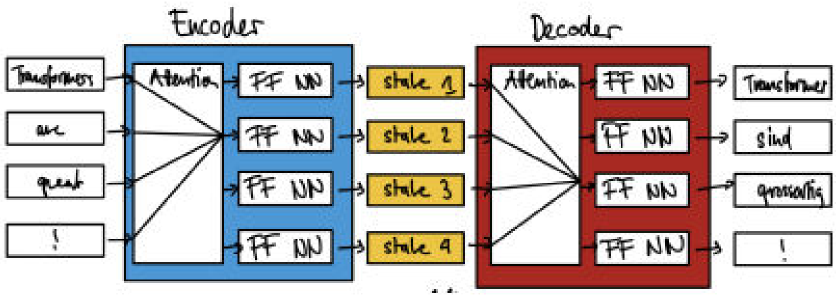
Chapter 1: Hello Transformers

- Transformers were designed to overcome the **bottleneck of RNNs**
- **General Mechanism of Attention in 2 parts**
  - **First:** allowing the **decoder** have **access** to **all encoder hidden states**
  - **Second:** decoder **prioritises hidden states** so the decoders only focus on what is important
- Each encoder outputs a **hidden state at each step** that the decoder can access
- The **decoder** assigns **weights** for each encoder that tells it how much **attention** to pay to that **state** when decoding the **next element** in the output



Self-Attention

- All of the tokens are fed **in parallel** into **self-attention layers** that are fed into **fully connected networks**
  - *Results in large computation efficiency gain*
- **Self-attention** creates a **different representation** of each **token** that is **dependent on surrounding tokens**
  - The **meaning of words changes** all the time
  - *Allows you to distinguish between apple the fruit and apple the company*

Transfer Learning

- A technique often used to train **convolutional neural networks** in **computer vision**
- You train the **first** network on **one broad task**, then **fine-tune** the **second** network on a **specialised task**
- Architecturally, you have a **body** and a **head**, head being the **task-specific network**

ULMFiT Training Process

- **Pretraining (Language Modelling) -** predict **next word** based on previous words
  - **No labelled data necessary** so you use a large corpus
- **Domain Adaptation -** After pretraining, adapt it to **in-domain corpus** by **fine-tuning** the **language model** weights
- **Fine-Tuning -** Fine tune the language model with a **classification layer for target task**

GPT and BERT - both transformers that make use of **pretraining** and **transfer learning**

- **GPT -** only uses transformer **decoders**
- **BERT -** only uses transformer **encoders** with **masked language modelling**
  - **Masked Language Modelling -** predict randomly masked tokens in text

Jax - Google's high-performance maths library for machine learning research

- **Automatically differentiate native Python and Numpy functions**

Hugging Face Transformers library - an API that combines PyTorch, TensorFlow and Jax

- Provides **task-specific heads** for easy fine-tuning of transformers

Sentiment Analysis - saying whether a piece of text is **positive** or **negative**

- A part of **Text Classification**

A Tour of Transformer Applications

- **Text Classification -** such as **sentiment analysis**
- **Named Entity Recognition (NER) -** recognising named entities in text (finding out what a piece of text is about)
    - **Named Entity -** names of products, places, people, etc.
- **Question Answering -** Give the model a **context** (passage of text), along with a **question**
    - **Extractive Question Answering -** Model returns span of text with the answer (**start and finish**)
        - Answer extracted directly from the text
- **Summarisation -** take long text and generate a **short version** with all relevant facts
    - Generation of text is **more complicated**
- **Translation -** Generates text in a different language
- **Text Generation -** Generate text after starting it of with some words

pipelines - Abstracts away all of the steps needed to convert text into predictions from a fine-tuned model

- `pipeline("sentiment-analysis")`
- `pipeline("ner", aggregation_strategy="simple")`
    - Outputs a score, an **entity_group** (org, loc, person, misc)
    - **Aggregation strategy -** group words according to predictions (e.g. two words given one category)
- `pipeline("question-answering")`
- `pipeline("summarization")`
- `pipeline("translation_en_to_de", model="Helsinki-NLP/opus-mt-en-de")`
- `pipeline("text-generation")`

Main Challenges With Transformers

- **Language Barrier -** research dominated by English
    - Difficult to find pre-trained models for some languages
    - **Potential Solution - zero-shot cross-lingual transfer**
- **Data Hungry -** transformers have lots of parameters that require a lot of data to train
    - **Potential Solution -** semi-supervised / unsupervised training
- **Black Boxes -** transformers are black boxes we don't understand
- **Biases -** transformers become biased to the data that we give it
    - Have to make sure text isn't racist, sexist, etc

Chapter 2: Text Classification

Maximum Context Size - (BERT: 512 tokens)

- The maximum input sequence length that can be accepted by a network
- Input can be made shorter by using an **end token**

Tokenisation - Converting raw strings into **numerical vectors of fixed size**

- It also involves breaking down words into **atomic units** (into sub-units learned from corpus)
- **Numericalization -** We give each token a **unique integer**
- **One hot encoding -** we convert a token's **unique integer** into a **one hot vector**
  - *The vector will have length of the vocabulary*
  - **`torch.nn.functional.one_hot(input)`**
- **Types of tokenisation**
  - **Character Tokenizer -** feeds each character individually into the model
    - Rarely used because it **ignores any structure in text such as words**
  - **Word Tokenizer -** split text into words
    - Punctuation is ignored
    - **Stemming -** converting "greater", "greatest" -> "great"
    - Creates a large vocabulary is there's tons of unique words and extra rules in text
  - We often **limit the vocabulary size**
    - Often is **100K most common words** in corpus
    - **UNK token**
  - **Subword Tokenizer -** best of character and word tokenization
    - Use characters for rare character combinations / misspellings
    - **Learned** from the corpus used for pre-training
    - **Implementations -** Byte-Pair-Encoding, WordPiece, Unigram, SentencePiece
    - **Key Ideas**
      - **Simple Tokenisation -** text corpus split into words by whitespace / punctuation
      - **Counting -** all words in corpus counted
      - **Splitting -** words in tally split into subwords (*initially characters*)
      - **Subword Pairs Counting -** Subword pairs counter
      - **Merging -** Using **rules**, some **subword pairs are merged in corpus**
      - **Stopping -** Process stopped when vocabulary size limit reached

Training a Text Classifier page 36/417

-

Token Embedding - the input **first gets tokenised**, then each word is represented as a fixed size vector (in BERT, **768**)

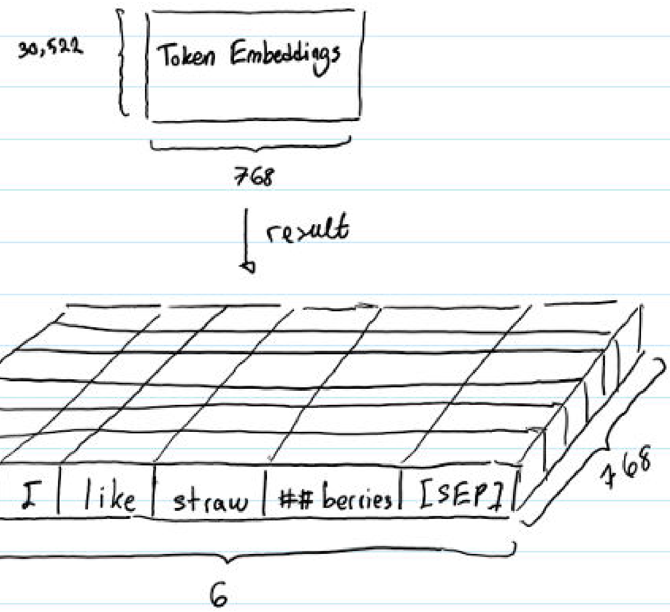
- This means that if the input sequence has 10 words, you get the **token encoding**, which means converting it into **10*500k** one hot vectors (represented as a **10*1 argmax**, then convert them into a **10*768 embedding**
  - **Token encoding** is also known as **sparse encoding/embedding**

" I like strawberries " , 3 words

↓ ①

"[CLS]", " I ", "like", "straw", "##berries", "[SEP]" , 6 tokens

↓ ②



$\overbrace{\text{"I like cats"}}^{\text{input 1}}$ $\overbrace{\text{"I like dogs"}}^{\text{input 2}}$, 2 inputs

↓ ① concat and tokenize

[CLS] I like cast [SEP] I like dogs , 8 tokens

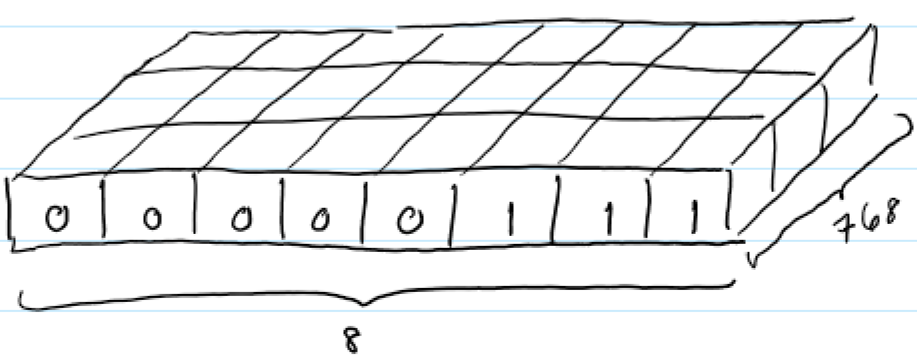
↓ ② label to distinguish input

[CLS] I like cast [SEP] I like dogs , 0 : input 1
0 0 0 0 0 1 1 1   1 : input 2

↓ ③ Lookup vector representation

$2 \left\{ \underbrace{\boxed{\text{Segment Embeddings}}}_{768} \right.$

↓ result



- The **token embeddings** have a shape of **n*768**
- The **segment embeddings** (which sentence does each word belong to) has a shape of **n*768**
- The **position embeddings** have a shape of **n*768**
- They **all get summed** element-wise to produce a single **n*768** vector passed into the encoder
- These are also known as **dense embeddings**

Self-Attention

- Takes in **token embeddings** $x_1, \ldots, x_n$ and converts them into a **contextualised embedding** $y_1, \ldots, y_n$ where each ***y*** is computed using **all *x*'s**
  - $$y_i = \sum_{j=1}^{n} w_{ji} x_j \,.$$
- **Scale Dot-Product Attention**
  - Each **token embedding** is projected into three vectors **query**, **key**, and **value**
  - Computing **attention scores**
    - First determine how **similar** the **query** and **key** are with a **similarity function** (i.e. **dot product**)
      - **Attention scores - query . key** (**n*n** matrix)
  - Computing **attention weights**
    - Attention **scores** are multiplying by a **scaling factor**, then **normalised** with **softmax**
      - *For numerical stability*
      - Usually **scaled by size of embedding vector (sqrt(768)**
    - Results in **attention weights** which have a dimension of **n*n**
    - **Attention weights w_{ij}**, n*n
  - **Update** token **embeddings**
    - We multiply the **values v** with the new **attention weights w**
    - $$y_i = \sum_j w_{ji} v_j.$$

Multi-headed attention

- Queries, keys and values are all unique **linear projections** of our **initial token vector**
- **Token -** can be words or image patches
- **Query -** what the token wants to **know about the other tokens**
- **Key -** what the token **contains**
- *Each query is compared to each key, and then the information **Value** is routed depending on the size of the dot p*
-