## Dealing with textual data

* Textual data is unstructured: Unlike numeric data, textual data is unstructured and cannot be directly used as input to a neural network. Before we can use RNNs for tasks like sentiment analysis or question answering, we need to preprocess the text data and convert it into a structured format.
* Vocabulary size: Textual data typically has a larger vocabulary size compared to numeric data. For example, the English language has over 170,000 words. To process text data, we need to convert words into a numerical representation that can be understood by the neural network.
* Handling variable-length input: Textual data is variable-length, meaning that different text inputs can have different lengths. This makes it challenging to use traditional neural networks that require fixed input sizes. To handle variable-length inputs, we need to use techniques like padding to ensure that all inputs are of the same length.
* Semantic representation: Unlike numeric data, where each value has a clear meaning, words in text data can have different meanings depending on their context. For example, the word "bank" could refer to a financial institution or the edge of a river. To ensure that the neural network can understand the meaning of words in a given context, we need to use techniques like word embeddings that represent each word as a vector of continuous values that capture its semantic meaning.

By learning techniques like tokenization, stopword removal, stemming and lemmatization, encoding techniques like bag of words, TF-IDF, and word embeddings, and padding, you would be able to preprocess textual data and use RNNs to perform tasks like sentiment analysis, question answering, text generation, etc.

## Word Embeddings

Word embeddings are a type of distributed representation used in natural language processing (NLP) that allow words to be represented as dense vectors of real numbers. Each word is mapped to a unique vector, and the vector space is designed such that words that are semantically similar are located close to each other in the vector space.

Word embeddings are typically learned through unsupervised learning techniques, such as neural network models like [Word2Vec](https://arxiv.org/pdf/1301.3781.pdf) and [GloVe](https://nlp.stanford.edu/pubs/glove.pdf" \t "_blank), which are trained on large corpora of text. During training, the model learns to predict the context in which a word appears, such as the surrounding words in a sentence, and uses this information to assign a vector representation to each word.

Pytorch provides several models for word embedding, including:

* **GloVe** (Global Vectors): It is a method for generating word embeddings, which are dense vector representations of words that capture their semantic meaning. The main idea behind GloVe is to use co-occurrence statistics to generate embeddings that reflect the words' semantic relationships. GloVe embeddings are generated by factorizing a co-occurrence matrix. The co-occurrence matrix is a square matrix where each row and column represents a word in the vocabulary, and the cell at position (i, j) represents the number of times word i and word j appear together in a context window. The context window is a fixed-size window of words surrounding the target word. The factorization of the co-occurrence matrix results in two smaller matrices: one representing the words, and the other representing the contexts. Each row of the word matrix represents a word in the vocabulary, and the entries in that row are the weights assigned to each dimension of the embedding. Similarly, each row of the context matrix represents a context word, and the entries in that row are the weights assigned to each dimension of the context embedding. The GloVe embeddings are computed by multiplying the word and context embeddings together and summing them up. This produces a single scalar value that represents the strength of the relationship between the two words. The resulting scalar is used as the value of the (i, j) entry in the word-context co-occurrence matrix. In PyTorch, you can use the torchtext package to load pre-trained GloVe embeddings. The torchtext.vocab.GloVe class allows you to specify the dimensionality of the embeddings (e.g. 50, 100, 200, or 300), and the pre-trained embeddings are downloaded automatically.
* **FastText**: FastText is a popular method for generating word embeddings that extends the concept of word embeddings to subword units, rather than just whole words. The main idea behind FastText is to represent each word as a bag of character n-grams, which are contiguous sequences of n characters. FastText embeddings are generated by training a shallow neural network on the subword units of the corpus. The input to the network is a bag of character n-grams for each word in the vocabulary, and the output is a dense vector representation of the word. During training, the network uses a negative sampling objective to learn the embeddings. The objective is to predict whether or not a given word is in the context of a target word. The model learns to predict the context of a word by computing the dot product between the target word's embedding and the embedding of each subword unit in the context. FastText embeddings have several advantages over traditional word embeddings. For example, they can handle out-of-vocabulary words, as long as their character n-grams are present in the training corpus. They can also capture morphological information and handle misspellings, since they are based on subword units. In PyTorch, you can use the torchtext package to load pre-trained FastText embeddings. The torchtext.vocab.FastText class allows you to specify the language and the dimensionality of the embeddings (e.g. 300).
* **CharNgram**: It refers to a method of generating character-level embeddings for words. The idea behind charNgram is to represent each word as a sequence of character n-grams (substrings of length n), and then use these n-grams to generate a fixed-length embedding for the word. For example, if we use CharNGram with n=3, the word "hello" would be represented as a sequence of 3-character n-grams: "hel", "ell", "llo". We would then use these n-grams to generate a fixed-length embedding for the word "hello". This embedding would be a concatenation of the embeddings of each n-gram. The benefit of using charNgram embeddings is that they can capture information about the morphology of words (i.e. how the word is formed from its constituent parts), which can be useful for certain NLP tasks. However, charNgram embeddings may not work as well for tasks that rely heavily on semantic meaning, since they do not capture the full meaning of a word. In PyTorch, you can generate charNgram embeddings using the torchtext package. The torchtext.vocab.CharNGram class allows you to generate character n-grams for a given text corpus, and the resulting n-grams can be used to generate charNgram embeddings for individual words.
* **BERT**: a pre-trained language model that can be fine-tuned for various downstream NLP tasks and also produces high-quality word embeddings. Pytorch provides pre-trained BERT models through the [transformers](https://pytorch.org/hub/huggingface_pytorch-transformers/" \t "_blank) library.

# Transformers

## Intro

Transformer models are a type of neural network architecture. They are designed to process sequential data (e.g., words in a sentence), such as natural language text. But here is why transformer models are revolutionary - they use a self-attention mechanism.

This self-attention mechanism allows them to focus on different parts of the input sequence and adjust their importance when making predictions about the output. In contrast Recurring Neural Networks (RNNs)/ Long Short-Term Memory (LSTM)/ Gated recurrent units (GRUs) are other types of Neural Networks that process a sequence one element at a time. Unlike self-attention, RNNs process the sequence in a linear fashion, with each element being processed sequentially based on its position in the sequence. As a result, these have a limited attention span and cannot “remember” the context from an earlier part of the sequence or conversation. Let’s see this with a visual.

So while LSTMs have been very effective in handling sequential data, they do have some limitations:

* Limited attention span - They struggle to capture long term dependencies in sequences as they maintain a limited amount of information in memory.
* Computation efficiency - LSTMs are computationally expensive to train.
* Handling multiple sequences - LSTMs are designed to handle one sequence at a time.

Transformers overcome all these limitations of LSTM by using self-attention and parallel processing.

The availability of these powerful transformer models can be found in numerous open-source APIs are currently accessible from various companies, including OpenAI, TensorFlow Hub, AWS, Google Cloud AI Platform, and Hugging Face Transformers. These APIs offer convenient integration into the data pipelines of businesses, allowing them to take advantage of pre-existing transformer models in deep learning and data science.

## Transformers in detail

Diagrama

Descripción generada automáticamente

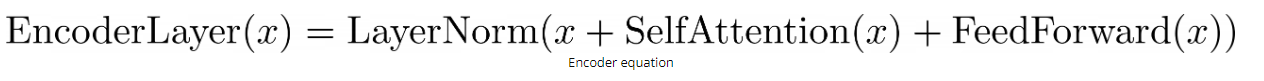
At a high level, the transformer architecture consists of an encoder and a decoder.

* The encoder takes in a sequence of input tokens and produces a sequence of hidden representations
* The decoder takes in the encoder's output and generates a sequence of output tokens.

The key innovation of transformers is the use of self-attention mechanisms, which allow the model to selectively focus on different parts of the input sequence when computing the hidden representations.

The self-attention mechanism works by computing attention weights between each input token and all other input tokens and using these weights to compute a weighted sum of the input token embeddings. The attention weights are computed using a softmax function applied to the dot product of a query vector, a key vector, and a scaling factor. The query vector is derived from the previous layer's hidden representation, while the key and value vectors are derived from the input embeddings. The resulting weighted sum is fed into a multi-layer perceptron (MLP) to produce the next layer's hidden representation.

More specifically, given an input sequence of length L, the encoder can be represented by a series of L identical layers, each consisting of a self-attention mechanism and a feedforward neural network:



Veámoslo gráficamente:

1. Tenemos 3 palabras. Este ejemplo solo se va a centrar en la segunda palabra (por eso solo tenemos 1 query). Esto se debería hacer para cada palabra.
2. Hacemos el embedding para convertirlas a vectores.
3. Necesitamos Query, Key y Value de cada embedding. De donde salen?

Multiplicando el embedding por Query/Key/Value matrix pasandolo por una Query/Key/Value layer que es una FFNN.

1. Obtenemos el score, que es multiplicar la Query por el Key.
2. Escalamos dividiendo el score por la raíz cuadrada de la dimensión del embeding, que vemos que es 4.
3. Se lo pasamos a una función softmax.
4. Multiplicamos el softmax score con el Value para obtenir el nivel de expresion de cada uno de estos vectores.
5. Se suman y se obtiene el self-attention context vector.

Diagrama

Descripción generada automáticamenteDiagrama

Descripción generada automáticamente

Imagen de la pantalla de un video juego

Descripción generada automáticamente con confianza media

Imagen de la pantalla de un video juego

Descripción generada automáticamente con confianza media

La ecuación sería:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

The key, value, and query vectors are used in the self-attention mechanism to help the model selectively attend to different parts of the input sequence.

* Key: You can think of the key vectors as a set of reference points the model uses to decide which parts of the input sequence are important.
* Value: The value vectors are the actual information that the model associates with each key vector.
* Query: Query vectors are used to determine how much attention to give to each key-value pair.

Example: imagine you are trying to summarize a long article. The key vectors could represent the most important sentences or phrases in the article, while the value vectors could represent the actual content of those sentences. The query vectors would then be used to decide which of these key-value pairs are most relevant to the task of summarization.

The self-attention mechanism works by computing a dot product between the query vector and each key vector, which produces a set of attention weights that indicate how much attention to give to each value vector. The resulting weighted sum of the value vectors represents the attended information for that particular query.

The decoder is similar to the encoder but also includes an additional attention mechanism that allows it to attend to the encoder's output.

## Benefits of transformers

Overall, the transformer architecture has several advantages over previous NLP models.

* First, it is highly parallelizable, which makes it more **efficient to train** on modern hardware.
* Second, it does not rely on any explicit notion of sequence order, which **allows it to better capture long-term dependencies** in the input sequence.
* Finally, the attention mechanisms allow the model to selectively attend to different parts of the input sequence, which helps it handle tasks such as language translation where the input and output sequences may have different lengths. Obtiene mejores resultados.

## BERT

BERT (Bidirectional Encoder Representations from Transformers) is a Machine Learning (ML) model for natural language processing developed by Google in 2018. BERT is a versatile model that can handle a range of natural language processing (NLP) tasks.

### The Science Behind BERT: How it Learns and Processes Language

To achieve its remarkable performance, BERT utilizes the following components:

**Extensive training data**

BERT was trained on a colossal dataset of 3.3 billion words, which is one of the main factors that contributed to its success. Specifically, it was trained on two vast datasets: Wikipedia (about 2.5 billion words) and Google's BooksCorpus (about 800 million words).

**MLM (Masked Language Modeling)**

MLM is a technique used by BERT to learn about the relationships between words in a sentence. In this process, BERT is trained to predict what a masked word should be based on the other words in the sentence.

Texto, Carta

Descripción generada automáticamente

**NSP (Next Sentence Prediction)**

NSP is another technique used by BERT during pre-training to help it better understand the overall structure and flow of language. In this process, BERT is trained to predict whether two sentences are likely to appear together in a piece of text.

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamenteTexto

Descripción generada automáticamente

## Steps to finetune BERT

1. First, we need to import all the necessary packages. We will use the [datasets](https://pypi.org/project/datasets/) library to load data and functions to compute metrics. From HuggingFace's [transformers](https://pypi.org/project/transformers/" \t "_blank) package, we will import tokenizers, trainers, and models for sentence classification.

**from** datasets **import** load\_dataset

**from** transformers **import** AutoTokenizer

**from** transformers **import** AutoModelForSequenceClassification

**from** transformers **import** TrainingArguments, Trainer

**import** numpy **as** np

**from** datasets **import** load\_metric

1. Next, we will define some functions to compute our metrics and tokenize our sentences.

**def** **compute\_metrics**(eval\_pred):

logits, labels = eval\_pred

predictions = np.argmax(logits, axis=-1)

**return** metric.compute(predictions=predictions, references=labels)

**def** **tokenize\_function**(examples):

**return** tokenizer(examples["text"], padding="max\_length", truncation=True)

1. Now, we can load and preprocess our dataset. Remember that we will use the datasets package to load data. The datasets package has many inbuilt datasets available, and you can find a list [here](https://huggingface.co/datasets" \t "_blank).
2. The tokenizer we select needs to be the same as the model we are using. There are many pre-trained models available in transformers and you can find a list of them [here](https://huggingface.co/transformers/pretrained_models.html" \t "_blank). In the code below, you can see that I am using the bert-base-cased model. Once we have selected the model, we need to tokenize our dataset. I have also added code to use a small subset of the data to make training faster. However, you may choose to use the whole dataset by uncommenting the last two lines.

tokenizer = AutoTokenizer.from\_pretrained("bert-base-cased")

tokenized\_datasets = raw\_datasets.map(tokenize\_function, batched=True)

small\_train\_dataset = tokenized\_datasets["train"].shuffle(seed=42).select(range(1000))

small\_eval\_dataset = tokenized\_datasets["test"].shuffle(seed=42).select(range(1000))

*# full\_train\_dataset = tokenized\_datasets["train"]*

*# full\_eval\_dataset = tokenized\_datasets["test"]*

1. Now that we have written our data preprocessing code, we can download our model and start to train it. We will use the AutoModelForSequenceClassification API to fetch the pre-trained bert-base-cased model. We also need to specify the number of classes in our data.

Finally, we can train and evaluate the model using a Trainer object.

model = AutoModelForSequenceClassification.from\_pretrained("bert-base-cased", num\_labels=<your labels>)

metric = load\_metric("accuracy")

training\_args = TrainingArguments("test\_trainer", evaluation\_strategy="epoch")

trainer = Trainer(

model=model,

args=training\_args,

train\_dataset=small\_train\_dataset,

eval\_dataset=small\_eval\_dataset,

compute\_metrics=compute\_metrics,

)

trainer.train()

trainer.evaluate()

Note: Fine tuning BERT takes a long time (even on GPUs), hence we are not providing a workspace for this demo. Please try this on your local machine.

## GPT

GPT, or Generative Pre-trained Transformer, is an advanced [autoregressive](https://en.wikipedia.org/wiki/Autoregressive_model" \t "_blank) language model built on the transformer architecture, which leverages self-attention mechanisms for efficiently handling long-range dependencies in sequence data. The primary goal of GPT models is to predict the next token in a given sequence by learning a probability distribution over a vast vocabulary. This is achieved through unsupervised pre-training on large-scale text corpora, followed by fine-tuning on specific tasks to generate human-like text, perform translation, answer questions, and more.

The evolution of GPT began with GPT-1, which demonstrated the potential of unsupervised pre-training followed by task-specific fine-tuning. GPT-2, the successor, utilized a much larger dataset and model size, leading to substantially improved performance across various NLP tasks. However, its release was initially limited due to concerns about potential misuse. GPT-3 took the concept further, scaling up to 175 billion parameters and introducing the "[few-shot learning](https://paperswithcode.com/task/few-shot-learning" \t "_blank)" paradigm, which allowed the model to perform tasks with very limited task-specific training data.

GPT-4 builds upon the advancements of its predecessors, featuring an even larger model size and enhanced pre-training techniques. This latest iteration benefits from architectural improvements, such as sparsity and attention mechanisms that facilitate more efficient training and inference. GPT-4's greater capacity enables it to learn more sophisticated language patterns and generate higher-quality output across a broader range of tasks. Additionally, GPT-4 can be fine-tuned with smaller datasets, making it a powerful tool for specialized applications in various domains. Despite its impressive capabilities, GPT-4 still faces challenges in controlling generated content, ensuring factual accuracy, and mitigating biases present in training data.