Natural Language Processing Leveraging Entity-Aware Representations

José Malaquias
Instituto Superior Técnico

Abstract

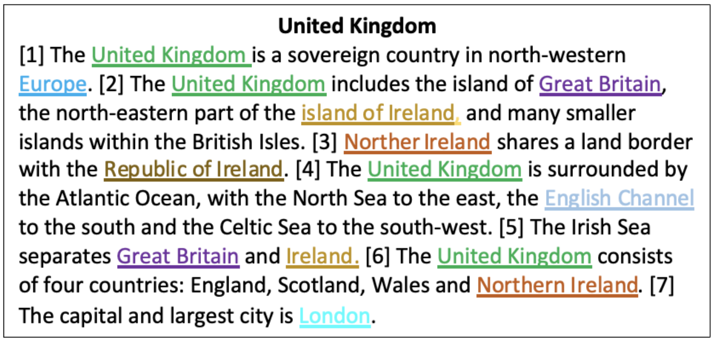
Recent pre-trained language models, based on self-attention and the Transformer neural architecture, produce a contextualized representation of words. Leveraging such representations in other neural networks has enabled achieving state-of-the-art results in the corresponding downstream Natural Language Processing tasks. A new line of research has focused on studying how to incorporate named entity components within a Pre-trained Language Model. The motivation is that named entities convey essential information that can further improve the performance of a model in a Natural Language Processing task when taken into account. However, current entity-focused models face limitations, such as the inability to process long inputs. In my M.Sc. research project, I will extend existing entity-aware self-attention models and propose techniques to solve the existing limitations in entity-related tasks.

1 Introduction

Recently, pre-trained language models (PLM) such as Bidirectional Encoder Representations from Transformers (BERT) (Devlin et al., 2019a), based on self-attention and the Transformer neural architecture, have outperformed previous approaches to modelling sequences of words within Natural Language Processing (NLP) applications, such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks. The self-attention mechanism proposed in the Transformer architecture (Vaswani et al., 2017) was a disruptive idea, as it allowed the modelling of dependencies without regard to their distance in the input or output sequences. Another disruptive idea was to treat words and entities in a given text as different types of tokens, (Yamada et al., 2020; Baldini Soares et al., 2019). Properly representing entities (Modrzejewski et al., 2020; Joko et al., 2021) is especially relevant, as many tasks (i.e., Question-Answering, Machine Translation, Chatbots) involve identifying the critical elements in a text, like names of people, places, or brands. Furthermore, their performance may benefit from the awareness of the representation of entities as it is mainly from those entities that we can locate the topic of a document. Specifically, we show an example in Figure 1 to understand why it is relevant to be aware of entities (highlighted words) and their relations within a text. In the example, in order to understand "Where is the Republic of Ireland" we have to consider the clues jointly: (1) The "United Kingdom" is located in "Europe" in sentence 1; (2) The "United Kingdom" includes part of the "island of Ireland" in sentence 2; (3) "Northern Ireland" shares a land border with "Republic of Ireland" in sentence 3; (4) "Northern Ireland" is one of the countries of "United Kingdom" in sentence 6. From the example we learn that in order to understand an entity we must consider its relations to other entities comprehensively. In the example, the entity "United Kingdom" appearing in sentences 1, 2, 4 and 6 help us find the answer. To understand "United Kingdom", we should consider all its connected entities and diverse relations among them, and to understand the relations between pairs of entities, we need to comprehend complex reasoning patterns in the text, i.e., we infer the "Republic of Ireland" is in "Europe" as it shares borders with a country of "United Kingdom", which is located in "Europe". It is through the relations between entities that we can capture the true meaning of an entity. For example, in sentence 5, with lack of context, the entity "English channel" could have multiple meanings. Suppose a model has a good comprehension of the context of a text. In that case, it will perform better on tasks that rely on the understanding, especially if we need to compare multiple relations along a multiple-sentenced document to apprehend such comprehension. However, most PLMs do not take into account the understanding of entities and their relations among multiple entities at the document level, whose context involves complex reasoning patterns. A recently proposed entity-aware model (Yamada et al., 2020) significantly improved the performance on entity-related tasks, such as entity typing, relation classification, named entity recognition, and others.

This M.Sc. research project focuses on the limitations of Language Understanding with Knowledge-based Embeddings (LUKE), a recently proposed approach to explicitly model entities (Yamada et al., 2020). I will focus on studying which aspects of its implementation can benefit from recent studies, making LUKE able to handle larger textual inputs (Beltagy et al., 2020; Zaheer et al., 2020) due to the quadratic cost of the attention mechanism; as well as acquiring a better understanding of the relations between entities (Qin et al., 2021; Kulkarni et al., 2021).

The remainder of this report is organized in the following manner: Section 2 summarizes the fundamental ideas associated with deep learning with Neural Networks (NN), as well as the foundations for comprehending the current state-of-the-art concepts in NLP. It also discusses relevant NN architectures for this research project. Section 3 introduces related work on the topic of entity-aware contextualized representations. Section 4 proposes the model to be developed and some of the techniques to be investigated, together with the



United Kingdom

[1] The United Kingdom is a sovereign country in north-western Europe. [2] The United Kingdom includes the island of Great Britain, the north-eastern part of the island of Ireland, and many smaller islands within the British Isles. [3] Norther Ireland shares a land border with the Republic of Ireland. [4] The United Kingdom is surrounded by the Atlantic Ocean, with the North Sea to the east, the English Channel to the south and the Celtic Sea to the south-west. [5] The Irish Sea separates Great Britain and Ireland. [6] The United Kingdom consists of four countries: England, Scotland, Wales and Northern Ireland. [7] The capital and largest city is London.

Figure 1: An example for a document "United Kingdom", in which all entities are underlined. We highlight the important entities and relations to find out "Where is the Republic of Ireland".

evaluation methods that I will apply. Section 5 concludes by summarizing the most important aspects of the research project.

2 Fundamental Concepts

This section describes critical concepts to understand the architectures presented in this report. It details how a basic NN works, how it can be used in machine learning tasks, and what techniques can be used to train and optimize NNs. Afterwards, two key models are explained: the Transformer architecture (Vaswani et al., 2017) and the BERT model (Devlin et al., 2019a).

2.1 Neural Models

Neural Networks (NNs) consist of artificial neurons (i.e., perceptrons), assembling a network capable of performing supervised learning. In theory, they are very good function approximators. Most NNs are *feed-forward*, meaning the data moves through them in one direction only, and is organized into layers of nodes. Nodes in the same layer do not connect: they receive data input from nodes in the previous layer and send its output to the next layer. Each connection has an assigned weight, multiplying with the input data. The products from the proceeding nodes are added to each other, resulting in a single number passed to an activation function. When training the NNs, the parameters (i.e., the weights) are adjusted until the output produced for data has consistent results with pre-existing annotations.

2.1.1 General Arquitecture

The simplest unit of a NN is the perceptron, corresponding to 2 single neurons and where the input x is fed directly to the output linearly via a series of weights, which can be formally defined as:

$$y = x \times W + b, \tag{1}$$

where y is the *pre-activated* output of the neuron, W represents the connection's weights, and b is a bias term. The bias allows the model to fit the prediction with the data better. Each weight's role is to define how much the dimension x_i stimulates the neuron responsible for y.

A single neuron can only learn strictly linear functions. Therefore, if we wish to tackle more complex problems, a possible solution is to stack several of these layers and feed the output of one layer as the input of the following layer. The layers placed between input and output layers are called *hidden layers*, and they increase the expressive power of the network. Formally, a Multi-Layer Perceptron (MLP) model can be expressed as:

$$\begin{aligned} y &= \left(x \times \mathbf{W}^1 + \mathbf{b}^1\right) + \mathbf{b}^2 \\ &= x\left(\mathbf{W}^1\mathbf{W}^2\right) + \left(\mathbf{b}^1\mathbf{W}^2 + \mathbf{b}^2\right) \\ &= x \times \mathbf{W} + \mathbf{b}, \end{aligned} \tag{2}$$

where $\mathbf{W}^n$ and $\mathbf{b}^n$ are respectively the weight matrix and bias vector of the layer l_n. Perceptrons can also be combined with an activation function, i.e. a function whose purpose is to introduce non-linearity into the output of a neuron. This helps the NN to learn complex patterns in data. Within a MLP, the output of a layer can be passed through an activation function that attenuates values below a certain threshold and increases values above it. Standard activation functions are the logistic sigmoid, the REctified Linear Unit (RELU), and the hyperbolic tangent (tanh). After applying an activation function, Equation 1 becomes:

$$y = g\left(x \times \mathbf{W} + \mathbf{b}\right), \tag{3}$$

where $g(z)$ is the desired activation function. Figure 2 represents a NN and its processes. In the figure, along with an input and an output layer, we have 2 hidden layers of perceptrons placed between them, which goal is to perform nonlinear transformations of the inputs entered into the network.

Training a NN involves optimizing its parameters (i.e., the connection weight θ) for a target task. In an optimization algorithm, we use *objective function* $J(\theta)$ to evaluate a candidate solution θ. When we search for a possible solution, we wish to minimize the error between expected and produced values. As such, the *objective function* is also referred to as *loss function.* The idea is to quantify the *loss* in the form of a single real number during the training phase to measure how well the NN predicts the output on the training set. A *loss function* is computed for a single training example/input, whereas a *cost function* is the average *loss* over the entire training dataset. A commonly used *cost function* for training neural networks in regression problems is the *Mean*

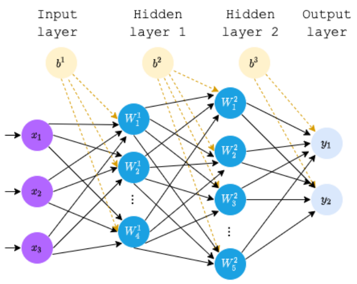


Figure 2: Graphical representation of a Multi-Layer Perceptron (feed-forward), with an input layer of dimensionality 3, 2 hidden layers of dimensionality 4 and 5, respectively, and an output layer of dimensionality 2.

Squared Error (MSE), which is formally expressed as:

$$J(\theta) = \frac{1}{2}\|f(\theta) - y\|^2, \tag{4}$$

, where $f(\theta)$ is the value returned by the model. In a NN, the loss function is only attached to the output layer, where predictions come out. To propagate the error to the weights and biases before the output, we make use of the *back-propagation algorithm* (Rumelhart et al., 1986).

Having established the concepts behind the architecture of a NN and acknowledging that calculating and minimizing a *loss function* for all values over the entire dataset can be challenging; we need a learning algorithm capable of finding the necessary parameter values (biases and weights). The mechanism responsible for the parameters' update is named Gradient Descent (GD).

2.1.2 Gradient Descent

GD is a popular optimization algorithm used to train machine learning models and NNs. GD can combine this strategy with multiple algorithms to find a local minimum of a differentiable function. In deep learning, GD is used to find the coefficients that minimize a function $J(\theta)$. Once the initial parameter values ($\theta \in \mathbb{R}^d$) are defined, GD uses calculus to iteratively update the values in the opposite direction of the gradient of the objective function $\Delta_\theta J(\theta)$, in order to minimize the *cost function*.

Mathematically, the gradient is a partial derivative concerning its inputs, which measures the change in weights concerning the change in the error. The gradient can be seen as the slope of a function. The slope is higher if the gradient is high, and the model can learn faster. On the other hand, if the slope is zero, the model cannot learn. For the parameters corresponding to weights

w and biases b, if we have the following cost function:

$$J(\theta) = J(w, b) = \frac{1}{N} \sum_{i=1}^{n} \left(y_i - (wx_i + b)\right)^2, \tag{5}$$

then the gradient is given by:

$$\Delta J(\theta) = \Delta J(w, b) = \begin{bmatrix} \frac{df}{dw} \\ \\ \frac{df}{db} \end{bmatrix} = \begin{bmatrix} \frac{1}{N} \sum -2x_i \left(y_i - (wx_i + b)\right) \\ \\ \frac{1}{N} \sum -2 \left(y_i - (wx_i + b)\right) \end{bmatrix}. \tag{6}$$

We iterate using the updated parameter values and compute the partial derivatives to solve this gradient through our data points. The values of the new gradient indicate the slope of our cost function at our current parameter values and the direction we should move to update the parameters. The learning rate *eta* gives the size of the steps we take to reach the local minimum is provided by the learning rate η.

Combining GD with the back-propagation algorithm, which is based on the chain rule, the calculation of the gradient flows backwards through the network, with the gradient of the final layer of weights being calculated first and the gradient of the first layer of weights being calculated last. Some computations regarding the gradient from one layer are reused in the previous layer. This backwards flow of the error information enables a more efficient calculation of the gradient at each layer by opposition to the naive approach of calculating the gradient of each layer separately.

The back-propagation algorithm starts with a regular forward pass, where the NN takes the input and sends it through all the layers until the output is produced. The prediction error is obtained by comparing the output fed to the loss function with the expected (ground truth) value. Finally, starting from the final layer, the error will be back-propagated inside the NN, updating the parameters of the nodes of each layer along the way.

The three main variants of GD differ on their accuracy, the time they take to perform an update, and also the amount of data they require to compute the gradient of the loss function.

Batch Gradient Descent is the simplest form of GD where all parameters are updated using the average of the gradients over all the training data. This approach is more straightforward and allows convergence to the global minimum in the majority of the cases. However, using all the training data in each epoch can be a drawback in cases where there is a lot of data. Each epoch has only one step of GD. This gradient is given by:

$$\Theta_{t+1} = \Theta_t - \eta \Delta_{\Theta} J(\Theta_t), \tag{7}$$

where Θ are the parameters of the model, weights and biases, $J\,(.)$ is the loss function, and η is the learning rate, which defines the size of the learning step.

Stochastic Gradient Descent is a variant where the computation of the gradient is done for each training example separately. It is commonly used because it is computationally the cheapest alternative. In each step, to update the parameters, we only use one instance of the dataset to estimate the gradient. Even though it consumes fewer resources, each update becomes more irregular, as the training example can vary. In this method, we need to tune the learning rate to achieve convergence carefully. Formally, this method is expressed in the following equation:

$$\Theta_{t+1} = \Theta_t - \eta \Delta_\Theta J(\Theta_t, x_i, y_i), \tag{8}$$

where x_i is a training example and y_i is its correspondent label.

Concerning the aforementioned methods, batch gradient descent is better suited for smoother curves and converges directly to minima, whereas stochastic gradient descent converges faster for larger datasets. As the stochastic gradient descent can only use one example at a time, it can slow down the overall computations.

Mini-Batch Gradient Descent is a variant that attempts to combine the advantages of both previous approaches. It splits the dataset into smaller batches of size $k << n$, and performs an update for every mini-batch of n training examples. Knowing $x_{i:i+n}$ represents the mini-batches and $y_{i:i+n}$ the labels for each training example within a mini-batch n, we have the following equation:

$$\Theta_{t+1} = \Theta_t - \eta \Delta_\Theta J(\Theta_t, x_{i:i+n}, y_{i:i+n}). \tag{9}$$

By using mini-batches, the variance of the parameter updates is reduced, leading to more stable convergence. This approach can also use highly optimized matrix optimizations, making computing the mini-batch gradient very efficient. The parameter k can be adjusted to meet the size of our data best, making it adaptable to the data and hardware. The size of the mini-batches usually ranges between 50 and 256.

The three variants all have their advantages as well as disadvantages. Usually, mini-batch gradient descent is more commonly used with NNs. However, other variants could be better suited to different situations or contexts.

2.1.3 Gradient Descent Optimization Algorithms

GD algorithms by themselves face some difficulties, such as the ability to select an appropriate learning rate or defining the tempo for updating the learning rate. If the learning rate is too low, it can lead to slow convergence, while a learning rate that is too large can retard convergence and cause the loss function to fluctuate around the minimum or even diverge. To solve these issues and others, different optimization algorithms have been proposed.

Momentum (Qian, 1999) is a method that helps accelerate stochastic gradient descent in the relevant direction, allowing it to escape local optima more easily. It also impacts the oscillation provoked in consecutive updates, decreasing the number of updates needed to converge. This is done by adding a fraction γ of the update vector of the previous time step to the present update vector. Knowing γ is the momentum term, and v_t is the vector of updated values in each iteration, the definition of momentum is expressed as follows:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \Delta_\Theta J\left(\Theta_{t-1}\right), \\ \Theta_t &= \Theta_{t-1} - v_t. \end{aligned} \tag{10}$$

Although there are many optimization algorithms in the literature (Ruder, 2017), such as Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop, AdaMax, Nadam or AMSGrad, we will only focus on Adaptive Moment Estimation (ADAM) introduced by Kingma and Ba Kingma and Ba (2015).

In brief, ADAM is a combination of momentum (Qian, 1999) and RMSprop (Tieleman et al., 2012), which computes adaptive learning rates for each parameter. It delivers good results alongside sparse gradients or a multitude of parameters. Similarly to momentum, Adam keeps an exponentially decaying average of past gradients m_t. Being v_t the decaying average of past squared gradients (uncentered variance), m_t the average of squared gradients (mean) and β_1 and β_2 momentum terms, the estimates of the first and second moments can be expressed by:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + \left(1-\beta_1\right) \Delta_\Theta J\left(\Theta_{t-1}\right), \\ v_t &= \beta_2 v_{t-1} + \left(1-\beta_2\right) \left(\Delta_\Theta J\left(\Theta_{t-1}\right)\right)^2. \end{aligned} \tag{11}$$

The parameters m_t and v_t are initialized as vectors of zeros, which cause them to be biased towards zero, especially during the initial time steps or when the decay rates are small (i.e., when β_1 and β_2 are close to zero). To counterbalance the biases, bias-corrected first and second moment estimates are computed:

$$\begin{aligned} \hat{m_t} &= \frac{m_t}{1-\beta_1^t}, \\ \hat{v_t} &= \frac{v_t}{1-\beta_2^t}. \end{aligned} \tag{12}$$

To update the parameters, the Adam update rule is applied using the aforementioned moment estimates:

$$\Theta_{t+1} = \Theta_t - \frac{\eta}{\sqrt{\hat{v_t}} + \epsilon} \hat{m_t}, \tag{13}$$

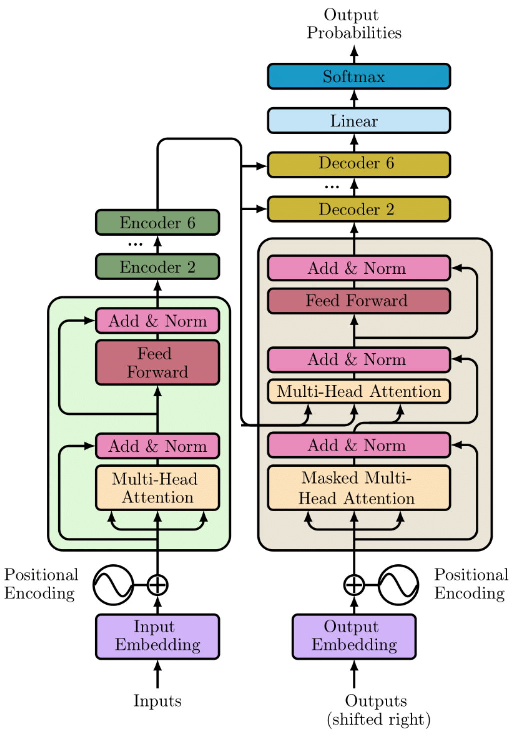


Figure 3: The Transformer model architecture.

where the parameter ϵ stands for a small scalar to prevent division by zero. In practice, Adam works well, achieving better results than other adaptive learning-method algorithms.

2.2 The Transformer

The Transformer (Vaswani et al., 2017) is a recent approach that simplifies tasks related to Neural Architecture sequence-to-sequence processing (i.e., processing sequences of words), which consists of transforming one sequence into another one with the help of two elements: an encoder and a decoder. The model uses an *attention* mechanism that takes into account, as an input, the relationships between all words in an input sentence, providing context for any position in the input sequence.

The Transformer architecture can be seen in detail in Figure 3 - The figure shows each of its components, namely the encoder and decoder layers, each attention mechanism, and the Feed-Forward Neural Network (FFNN) blocks. In the following subsections, we will unfold each component.

2.2.1 Multi-Head Self-Attention

The Transformer uses a scaled *dot-product* attention. This mechanism uses three vectors, respectively called query $\mathbf{q}$, key $\mathbf{k}$, and value $\mathbf{v}$. To obtain them, each vector in the input sequence, which we denote in Tensor form as $\mathbf{X}$, has to be multiplied by each vector's weight matrix learnt while training.

$$\begin{aligned}\mathbf{Q} &= \mathbf{X}\mathbf{W_q},\\ \mathbf{K} &= \mathbf{X}\mathbf{W_k},\\ \mathbf{V} &= \mathbf{X}\mathbf{W_v}.\end{aligned} \tag{14}$$

The role of $\mathbf{Q}$ is to hold the current words being analyzed, while $\mathbf{K}$ has an indexing mechanism for the value vectors and $\mathbf{V}$ saves information about each input word. The next step is to create self-attention for every word in the input sequence. We pick one word in a sentence and calculate the score for all words concerning the chosen word. This result gives the importance of all words to the word in question. The score is given by the *dot-product* of the query vector with each one of the key factors. The closest query-key product will have the highest value, meaning the words associated with the pair are more similar. These scores are then divided by the square root of the dimension key factor to having more stable gradients. To normalize the scores, one applies a *softmax* function to guarantee they are all positive and add up to 1. The *softmax* function also removes low values by driving them towards low probability scores.

$$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \tag{15}$$

where $\mathbf{z}$ is a vector with dimension K and $i = 1, ..., K$.

Finally, each *softmax* is multiplied to the corresponding value vector v to eliminate irrelevant words. All these results are summed, forming the output of the self-attention layer, i.e. a $\mathbf{z}$ vector of size d_k, which will be passed to the Feed-Forward Network (FFN) as an input. The idea is to have a $\mathbf{z}$ vector for each word in a sentence and, in tensor form, we have:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{W}^T}{\sqrt{d_k}}\right)\mathbf{V}. \tag{16}$$

Each word has an attention vector assigned to itself. However, one attention vector may not be enough to capture multiple possible interactions. To solve this, we take 8 attention vectors per word and use a weighted average to compute the final attention vector for every word. Since the self-attention operation is calculated multiple times, in parallel and independently, this process is commonly known as multi-head attention.

The main advantage of the multi-head attention is training stability, since this way we can use a smaller number of layers than with single-head attention.

Attending the same number of positions enables the model to have a greater power to encode multiple relationships and nuances for each word. Using h parallel attention layers, commonly known as heads, the multi-head attention mechanism allows the model to attend to information from various representation subspaces concurrently. To do so, we use distinct projected $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ values in each head to calculate attention. Consequently, we combine the findings and project the resulting concatenation into the original dimensions. Formally, this may be written as follows:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}\left(head_1, head_2, ..., head_h\right)\mathbf{W}^O, \tag{17}$$

where

$$\text{head}_i = \text{Attention}\left(\mathbf{Q}\mathbf{W}_i^{\mathbf{Q}}, \mathbf{K}\mathbf{W}_i^{\mathbf{K}}, \mathbf{V}\mathbf{W}_i^{\mathbf{V}}\right). \tag{18}$$

The letter h stands for the number of attention heads, whereas $\mathbf{W}_i^{\mathbf{Q}}$, $\mathbf{W}_i^{\mathbf{K}}$, $\mathbf{W}_i^{\mathbf{V}}$ and $\mathbf{W}^O$ are parameter matrices.

2.2.2 Input for the Model

Before the inputs go inside the encoder/decoder layers to be processed, we have to change their representation using embeddings and positional encoding. Word embeddings are word representations that allow words with similar meanings to have exact representations, capturing their meaning. They help capture the semantic or context and help to understand the relationship between terms in the sentence. To generate embeddings, we apply feature extraction-based techniques to map words or phrases from the vocabulary to vectors of real numbers, which can be later used in computations. For the Transformer, each token of our input sequence will be transformed into a vector of size $d = 512$. This size is kept constant throughout all steps of the model. A word embedding layer can be explained as a table where the model grabs a learned vector representation for each word of a phrase.

The self-attention mechanism does not consider the order of the input sequence, which is an essential aspect of many NLP tasks. Therefore, there is the need to implement a mechanism that attributes to each token a relative position. This is achieved by adding a position encoding vector to each input embedding. This is only done before the word embeddings are fed into the model. The encoding vectors are not learnt as part of the model, and instead, they are hard-coded using the following function:

$$\begin{aligned} \text{PE}(pos, 2i) &= \sin\left(\frac{pos}{1000^{\frac{2i}{d}}}\right), \\ \text{PE}(pos, 2i+1) &= \cos\left(\frac{pos}{1000^{\frac{2i}{d}}}\right), \end{aligned} \tag{19}$$

where pos is the token's position in the text sequence and i the vector dimension, so that even and odd dimensions get different positional encodings. Each dimension of the position encoding corresponds to a sinusoid.

The embedding and position encoding of the input sentence make every token unique. Two properties encoders inform the model: the absolute position of each word in the input sentence and the relative distance between different words in the sentence.

2.2.3 The Encoder Component

In a Transformer, the encoder is formed by 6 smaller sequential encoder layers, each containing the same structure but different parameters. Each encoder contains two sub-layers, namely a multi-head attention layer whose goal is understanding how neighbour positions affect the current one, followed by a fully-connected FFNN layer, sharing the same input and output formats.

The position-wise Feed-Forward Networks (FFN) is present in both encoder and decoder layers after each multi-head attention sub-layer. This NN is not incorporating information from other words. It looks at only one word at a time, in contrast with the attention layer. This consists of two linear transformations with a RELU activation among them. The input and output layers have size $d_{model} = 512$, whilst the inner layer has 4 times the size of the model $d_{ff} = 2048$.

There is also a residual connection (He et al., 2016) around each sub-layer, together with a normalization operation that normalizes each layer's result. Assuming x_1 is one vector of embeddings received by an encoder, and x_3 is the output of one of the 6 smaller encoders, the normalization operation can be formally expressed as follows:

$$\begin{aligned} x_2 &= \text{LayerNorm}\left(x_1 + \text{MultiHead}(x_1)\right), \\ x_3 &= \text{LayerNorm}\left(x_2 + \text{FFNN}(x_2)\right). \end{aligned} \tag{20}$$

The multi-head attention was explained in Section 2.2.1, whereas the normalized layer adds the input x to the implemented sub-layer function output. The way the LayerNorm() (Ba et al., 2016) operator is applied to the embeddings ν is formulated in the following way:

$$\text{LayerNorm}(\nu) = \gamma \frac{\nu - \mu}{\sigma} + \beta, \tag{21}$$

where μ and σ are the mean and standard deviation of the elements in ν, and where γ, β are hyperparameters associated with scale and bias vector, respectively. The output of the encoder is the following small encoder and the multi-head attention layer of the decoder.

2.2.4 The Decoder Component

The decoder is composed of a stack of 6 identical layers, as it happens in the encoder. Each decoder layer has a similar structure to each encoder layer. The main difference is the addition of a masked multi-head attention layer in the beginning. The decoder's inputs are embedded and position encoded the same way it happens to the encoder's input. The resulting vector of the embedding and encoding is fed into the masked-headed self-attention sub-layer. The term *masked* in this type of attention symbolizes that the predictions for position j can only depend on the known outputs of positions less than j.

The non-masked attention layer of the decoder, also known as the encoder-decoder attention layer, receives inputs from either the decoder's masked layer or the encoder's output. This configuration helps the decoder focus on appropriate parts of the input sequence. The encoder-decoder's output sequence prediction is achieved autoregressively, which means it is created iteratively, token by token. The output of the decoder stack is then fed back into the following decoder. Assuming x_1 is a embeddings vector, the architecture of a decoder is defined as follows:

$$\begin{aligned} x_2 &= \text{LayerNorm}\left(x_1 + \text{MaskedMultiHead}(x_1)\right), \\ x_3 &= \text{LayerNorm}\left(x_2 + \text{EncoderDecoderMultiHead}(x_2)\right), \\ x_4 &= \text{LayerNorm}\left(x_3 + \text{FFNN}(x_3)\right). \end{aligned} \tag{22}$$

The output embedding vector (x_4) is then passed to a linear layer, containing the output dimensionality of the vocabulary. Finally, the last operation is a *softmax* function, which computes the predicted probabilities for each word in the chosen vocabulary.

2.3 Bidirectional Encoder Representations from Transformers

The Bidirectional Encoder Representations from Transformers (BERT) (Devlin et al., 2019a) is a language model developed from the Transformer model. This model was a disruptive technology, as it had impressive benchmark performances in a set of different NLP tasks. The main idea is to use a Transformer encoder to produce deep bidirectional representations from the text that can be explored for target tasks. BERT produces contextual token representations in the sense that the same token can have a different representation vector depending on the surrounding context.

2.3.1 BERT Pre-Training Tasks

BERT is a PLM meaning it can be chosen from available models, where they are pre-trained over a large unlabeled generic corpus to perform a specific task. If suitable, the model can be used as the starting point for a model on the task of interest. However, one might wish to adapt a particular task or dataset over additional data; this is known as fine-tuning. From this approach, we can effectively adapt the model with generic language understanding capabilities to understand our dataset (i.e., apple may have 2 meanings: a fruit or a technology brand). BERT uses two pre-training tasks, namely Masked Language Model (MLM), and Next Sentence Prediction (NSP).

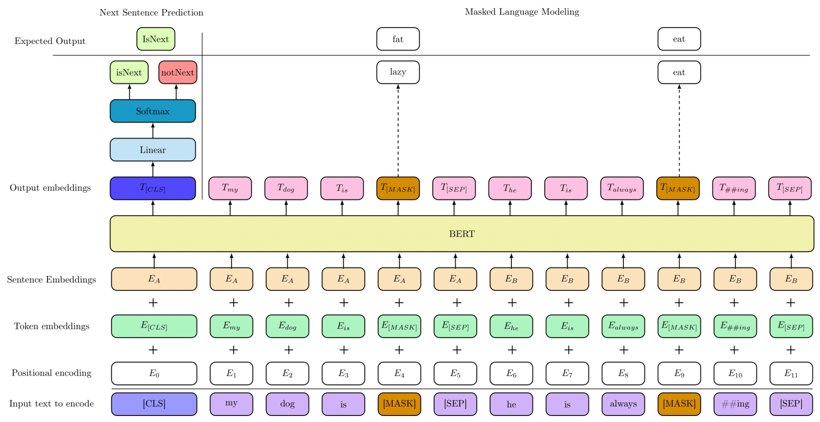

Next Sentence Prediction
Masked Language Modeling
Expected Output
IsNext
fat
eat
isNext
notNext
lazy
eat
Softmax
Linear
Output embeddings
$T_{[CLS]}$ T_{my} T_{dog} T_{is} $T_{[MASK]}$ $T_{[SEP]}$ T_{he} T_{is} T_{always} $T_{[MASK]}$ $T_{\#\#ing}$ $T_{[SEP]}$
BERT
Sentence Embeddings
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B E_B
Token embeddings
$E_{[CLS]}$ E_{my} E_{dog} E_{is} $E_{[MASK]}$ $E_{[SEP]}$ E_{he} E_{is} E_{always} $E_{[MASK]}$ $E_{\#\#ing}$ $E_{[SEP]}$
Positional encoding
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10} E_{11}
Input text to encode
[CLS] my dog is [MASK] [SEP] he is always [MASK] ##ing [SEP]

Figure 4: Input representation and pre-training tasks from BERT.

In the first task, displayed in Figure 4, we randomly mask a percentage of the input tokens (i.e., *fat, eat*), and try to predict them. In Figure 4, the masked tokens are fed into an output *softmax* over the vocabulary. Such operation enables us to obtain a bidirectional pre-trained model. However, there is a limitation, if we use this task for fine-tuning, the [MASK] token is not visible. A possible fix is not to assign a [MASK] token to every *masked* word, meaning only a small percentage of token positions are randomly assigned to be predicted, meaning each token has a defined probability of being either replaced by the [MASK] token, to be replaced by a random token, or not to be replaced. Finally, the final hidden vector for the token to be predicted is used to predict the original token with cross-entropy loss.

The second task displayed in Figure 4 is used in order for the model to understand the relationship between two sentences. The task consists of classifying whether sentence A is followed by sentence B. Each sentence has a label designated *isNext* or *notNext*. We assign sentence B as the actual next sentence 50% of the times when performing the task. In the remaining attempts, the *notNext* labelled sentence is randomly picked from the corpus. We compare the predicted output with the expected output and denote if their labels (*isNext*, *notNext*) match. Pre-training towards this task is very beneficial to Question-answering tasks.

After pre-training the model, we can modify BERT to perform certain tasks by replacing its output layers. In the original BERT implementation (Devlin et al., 2019a), the pre-training was done using the BooksCorpus (800 million words) and English Wikipedia (2500 million words) as datasets.

2.3.2 BERT Inputs

In the context of BERT, an input sequence is defined as an arbitrary span of contiguous text rather than an actual linguistic sentence. The BERT input representation was developed such that one or more sentences could be represented as a token sequence. The sequence of tokens has to go through some pre-processing steps before being fed into BERT: token embeddings (1), sentence embeddings, and (2) positional embeddings (3).

Every word token in the input sequence has to exist in BERT's fixed vocabulary, so every word has to be converted through the token word piece embedding (Zhang et al., 2019) into a vector representation, as Figure 4 shows. In the word embedding lookup table (vocabulary), there are about 30,000 words and 768 features, corresponding to the output vectors of BERT. Each token can define either a word, a set of characters, a symbol or a special mark. One example of a special mark is the [CLS] token, which is placed at the start of every input sentence, whereas [SEP] is used to separate segments of a sequence when there is more than one sentence (i.e., [SEP] separates the sentences "my dog is fat" and "he is always eating"). If a word does not exist in the vocabulary, it is split into smaller subwords. All subwords start with ## except for the first one (i.e., if the word eating does not exist in the vocabulary, it is tokenized to eat + ##ing). All sentences need to be truncated to a single fixed length, and the maximum allowed length is 512 tokens.

The token embeddings are then added to a segment embedding that identifies which sentence a token belongs to within a given sequence. If there is only one sentence, this embedding will be the same for every token.

Finally, a position embedding is also added. In the Transformer architecture (Vaswani et al., 2017), the encoding of the position was made through a hard-coded function. However, the BERT model uses learned embeddings, and it was trained with sequence length 128 for 90% of the steps, and later remaining 10% of steps were trained with a sequence length of 512 tokens in order to learn position embeddings. These embeddings are applied to determine the global position of any given token, and the relative distance among tokens.

For every token, its input representation is thus obtained by summing the corresponding token, segment, and position embeddings. All output vectors resulting from the embedding layer are unique.

2.3.3 The Overall BERT Architecture

The base implementation of BERT has an embedding layer followed by twelve modified Transformer encoder blocks. The size of the embedding vector was increased by 50% (512 + 256) to 768, and the number of attention heads increased to 12. The mechanism of each encoder sub-layer is the same as described in Section 2.2.3. The self-attention layer is shared among all tokens.

2.3.4 BERT Fine-Tuning

One of BERT's most significant advantages is the ability to adapt the model to multiple applications. The same model can be applied to multiple downstream tasks involving single texts or text pairs by adjusting the adequate inputs and outputs. As the number of parameters of BERT can range from 100 million to over 300 million, training the model from scratch on a small dataset would result in overfitting. We can consider 3 different fine-tuning techniques:

1. **Train the entire architecture**: In this case, the error is back-propagated through the entire architecture and the pre-trained weights are updated based upon the new dataset.
2. **Train some layers while freezing others**: keep the weights of the initial layers of the model frozen while we re-train only the higher layers. We can manually assign how many layers we wish to be frozen and how many to be trained.
3. **Freeze the entire architecture**: we can freeze all the layers of the main architecture and attach NN layers of our own, to train the new model. Only the weights of the attached layers would be updated during model training.

BERT by itself is not able to perform tasks; therefore, we have to adapt the model for the task we want to test. If we wish to do token-level tasks, we can feed the output token representations to the final classification part of the model (i.e., a *softmax* layer). On the other hand, we can provide the [CLS] token representation to a final classification head for sentence-level tasks.

2.3.5 BERT Variations

Recently, several researchers have proposed extensions of BERT with different configurations that deliver better performance in multiple areas.

Robustly optimized Bidirectional Encoder Representations from Transformers (RoBERTa) (Liu et al., 2019) was one of the first proposals, and the main idea was to improve BERT pre-training. Unlike BERT, RoBERTa uses dynamic masking over the training data to a different masking pattern every time a sequence is fed into the model. Furthermore, the authors also suggested that the original model was undertrained, and by altering some aspects, state-of-the-art results could be matched or surpassed in several datasets. Some alterations were made, such as increasing the number of training steps, training the model with larger batches over an increased amount of data, removing the next sentence prediction task, and training on longer sequences.

Improving Pre-training by Representing and Predicting Spans (SpanBERT) (Joshi et al., 2020) was aimed to improve the representation and prediction of spans of text in BERT. The authors proposed (1) to mask contiguous random spans instead of random tokens and (2) to train the span boundary representations to predict the entire content of the masked span without relying on the individual token representations within it. The Span-based masking enables the model to predict entire spans using only their context. This model outperformed BERT in all baselines, particularly when tasks involve contiguous spans of text.

mBERT (Devlin et al., 2019b) is an adaptation of BERT to languages other than English, simultaneously supporting 104 languages. For every language, all text was lowercased, the diacritics were removed, the punctuation was split, and whitespaces were tokenized. The general solution to produce a multilingual BERT was to pre-train on a mixture of many languages, learning knowledge shared between idioms and leveraging it for fine-tuning tasks of any language contained in the mixture. The strong contextual representations of BERT should make up for any ambiguity introduced by stripping accent markers.

Decoding-enhanced Bidirectional Encoder Representations from Transformers with Disentangled Attention (DeBERTa) (He et al., 2021) is another Transformer-based neural language model pre-trained on large amounts of raw text using self-supervised learning. The architecture improves BERT and RoBERTa by introducing two main innovations. The first idea is a disentangled attention mechanism that splits the input vector into two vectors: content and position. The attention weights are then computed using disentangled matrices over each one separately. The second innovation was to incorporate absolute positions in the Masked Language Modeling task by applying word position embeddings before the vectors enter the *softmax* layer. Furthermore, the author introduced a new virtual adversarial training method for fine-tuning the model to NLP tasks, which was proved to be effective at improving model generalization.

A Lite Bidirectional Encoder Representations for Self- supervised Learning of Language Representations (ALBERT) (Lan et al., 2020) introduced two parameter-reduction techniques to lower memory consumption and increase the training speed of BERT. The first one is a factorized embedding parameterization. The large vocabulary embedding matrix is decomposed into two smaller matrices, and the size of the hidden layers is separated from the size of the vocabulary embeddings. This alteration makes it easier to grow the hidden size without increasing the parameter size of the vocabulary embeddings. The second innovation corresponds to cross-layer parameter sharing. This prevents the parameters from growing with the depth of the network. By applying these techniques, the number of parameters for BERT was reduced, and the performance is not compromised. Comparing an ALBERT configuration to a similar BERT-large model, the first one has about eighteen times fewer parameters and can be trained about 1.7 times faster. The parameter reduction also stabilizes the training and helps with generalization.

2.3.6 Important Limitations in BERT

Experiments with different NLP tasks have shown that BERT implementations struggle with role-based event prediction and that they show apparent failures with the meaning of negation (Ettinger, 2020). It does not have sensitivity to the most probable answer in real-life situations. Also, BERT cannot deal with long input sequences. BERT only supports up to 512 tokens and, since the self-attention mechanism has computational and memory requirements that are quadratic, changing the supported token limit would introduce a significant increase in computational cost.

BERT also highly depends on the training dataset vocabulary. Suppose the model is performing tasks over dynamic areas of knowledge, where new

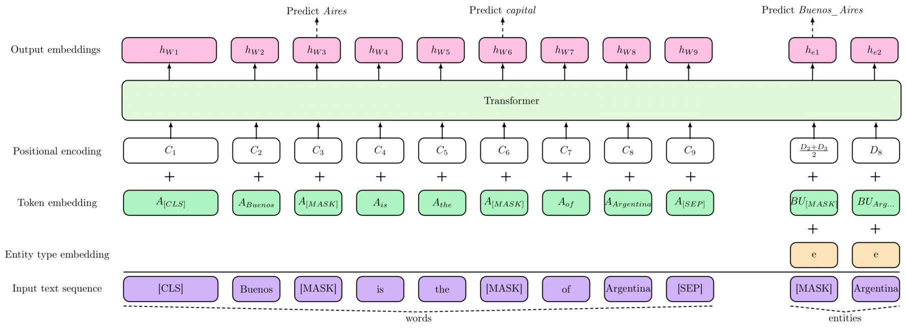


Figure 5: Input representation from LUKE.

words or types of writing are added from time to time. In that case, it will not recognize the unlabeled terms and will have a more data-scarce vocabulary on that field. A solution to deal with this is to perform continuous re-training and validation, but on top of being time-consuming, this is also costly and not practical.

3 Related Work

This section discusses published approaches that relate to the proposed work. NLP frequently involves tasks regarding entities (i.e., entity typing (ET), named entity recognition (NER), question answering (QA) and relation classification (RC), just to name a few). A good representation of entities is a decisive part of a good model and can be challenging.

Although transformer-based representations, i.e., produced by BERT and RoBERTa, correspond to good word representations, they are not very effective to represent entities for two reasons: (1) The computation of span-level representations of entities highly depends on the size of the downstream dataset. (2) Some entity-related tasks, such as RC and QA are built upon the relationship between pairs of entities. Even though the Transformer self-attention mechanism catches complex relationships between words, it cannot establish the relation between entities as many entities are composed of more than one word. They can be separated into multiple tokens in the model.

3.1 Language Understanding with Knowledge-based Embeddings

Language Understanding with Knowledge-based Embeddings (LUKE) (Yamada et al., 2020) is a language model based on RoBERTa (Liu et al., 2019) designed to improve the performance of entity related tasks. This approach adds entity embeddings as well as a new entity-aware self-attention mechanism. LUKE learns how to compute entity representations and delivers them in the model output.

3.1.1 Pre-training Task

The model is pre-trained through a standard masked language model (MLM) task together with a new pre-training task that focuses on learning dedicated entity-based representations. This is depicted in Figure 5, where it is possible to observe that the right side of the model's architecture focuses specifically on entities.

The model learns these representations by specifically targeting entities on a separate MLM task. To this end, the model must predict 15% of all words and entities during training. If an entity is not represented in LUKE vocabulary, it is assigned the [UNK] token to the referred entity. Finally, we predict the masked entity y.

The dataset used to train the novel task is extracted from a large amount of Wikipedia data, where the entities are the hyperlinks present in the Wikipedia website. The prediction of the masked entity $\hat{y}$ is formulated as follows:

$$\begin{aligned} m &= \text{Layer\_Norm}\left(\text{gelu}(\mathbf{W}_h\mathbf{h}_e + \mathbf{b}_h)\right), \\ \hat{y} &= \mathit{softmax}(\mathbf{BTm} + \mathbf{b}_0), \end{aligned} \tag{23}$$

assuming $\mathbf{h}_e$ is the Transformer layer output associated with the masked entity, $\mathbf{T}$ and $\mathbf{W}_h$ correspond to weight matrices, *softmax* and layer_norm correspond to the activation and layer normalization functions, respectively, B represents the entity token embedding matrix, and m stands for the number of words. The output of LUKE is a contextualized representation for each word and entity in the text.

3.1.2 Input Representation

A unique attribute of LUKE is the treatment of words and entities as independent tokens, as seen in Figure 5. The input representation of a token, which is either a word or an entity, is obtained by applying the following embeddings:

1. Token embedding: represents the type of token(i.e., A for words or B for entities);
2. Positional embedding: represents the token's absolute position in a word sequence. If an entity includes more than one token, its position embedding D is calculated by averaging the embeddings of the corresponding token positions C that form the entity;
3. Entity type embedding (e): means that the token is an entity.

The input representation of a word (i.e., left side of the input sequence in Figure 5) is given by adding the token and position embeddings, whereas to obtain the entity representation (i.e., *Buenos Aires*$_{[MASK]}$, *Argentina*) we add all the embeddings. As the masked input entity relates to multiple token entities (*i.e., Buenos Aires*), to encode its position, we average the corresponding position embeddings. For example, the entity *Buenos Aires* occupies positions 2 and 3 of the input word sequence, therefore to obtain the positional entity embedding, we have $\frac{D_2+D_3}{2}$. Similarly to previous model implementations (Devlin et al., 2019a; Liu et al., 2019), LUKE makes use of special tokens [CLS] and [SEP] to mark the first and last words of an input sequence.

3.1.3 Entity-Aware Self-Attention Mechanism

The idea behind the standard self-attention mechanism is to allow tokens to relate using attention scores. This is also implemented in LUKE with a few modifications. The traditional mechanism (Equation 16) does not expect two different kinds of tokens, while LUKE proposes a new attention mechanism that is aware of what kind of tokens it is dealing with when it is computing the attention score (e_{ij}) between the pair of tokens.

The aforementioned idea was implemented through an entity-aware query mechanism that handles query matrices with different weights for each possible combination of pairs of tokens x_i and x_j. The proposed attention mechanism is computed as follows:

$$e_{ij} = \begin{cases} \mathbf{K}x_j^T\mathbf{Q}x_i \text{ , if } x_i \text{ and } x_j \text{ are words} \\ \mathbf{K}x_j^T\mathbf{Q}_{w2e}x_i \text{ , if } x_i \text{ words and } x_j \text{ entity} \\ \mathbf{K}x_j^T\mathbf{Q}_{e2w}x_i \text{ , if } x_j \text{ words and } x_i \text{ entity} \\ \mathbf{K}x_j^T\mathbf{Q}_{e2e}x_i \text{ , if } x_i \text{ and } x_j \text{ are entities} \end{cases}, \tag{24}$$

where $\mathbf{Q}$, $\mathbf{Q}_{w2e}$, $\mathbf{Q}_{e2w}$ and $\mathbf{Q}_{e2e}$ are query matrices, and where $\mathbf{K}$ and $\mathbf{V}$ are respectively the key and value matrices.

3.1.4 Experiments

The model delivered state-of-the-art performances in 5 entity-related tasks with only one simple linear layer after the token representations.

- Entity typing: the goal is to predict the type of each entity (i.e., person, location, organization) in a given sentence.
- Relation classification: predict a relation between *head* and *tail* entities in a given sentence.
- Entity recognition: locate and classify named entities mentioned in unstructured text into pre-defined categories.
- Cloze-style QA: given a question and a passage of a text, identify the suitable entity.
- Extractive QA: predict the correct answer to a given question when a Wikipedia excerpt containing the answer is provided.

In Figure 6 we can see an example of how we can apply the representations LUKE gives us on a NLP task, particularly on the entity typing task. In this example, we mask the target entity and insert words and the entity in each sentence as the model's inputs. The target entity is classified using a linear classifier based on the corresponding entity representation. Like a multi-label

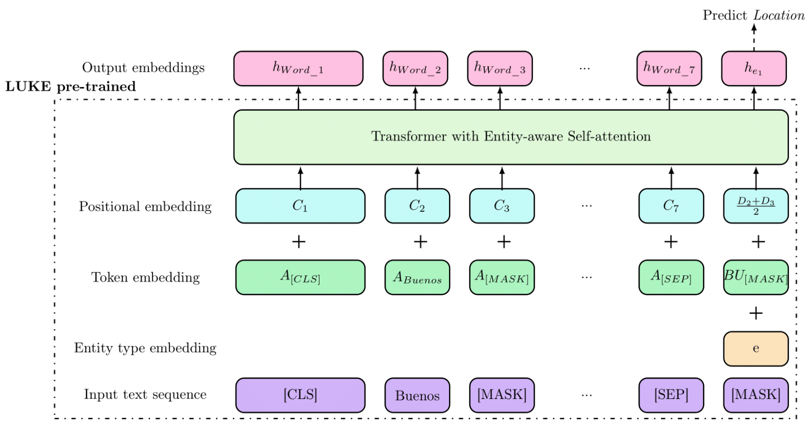


Figure 6: Entity typing task from LUKE.

classification task, the model is trained using binary cross-entropy loss averaged over all the 8 possible entity types to predict which type corresponds to the target entity. In the case of Figure 6, the entity type of the entity *Buenos Aires* is *Location*. The introduction of the entity-aware self-attention, and a personalized entity representation, enabled the model to outperform RoBERTa on every entity-related task. LUKE is effective at capturing relationships between entities.

3.1.5 Limitations

Although LUKE outperformed some models in important tasks, it also encountered some drawbacks. LUKE delivers good performance when the model receives both word and entity type tokens. As LUKE highly depends on datasets that label their entities, if a new entity that does not belong in the vocabulary is introduced, the model has to fall back to the [UNK] entity representation, damaging its performance in the target NLP task.

However, possible solutions could be to apply techniques referenced in Improving Entity and Relation Understanding for Pre-trained Language Models via Contrastive Learning (ERICA) (Qin et al., 2021).

Another LUKE limitation is the inability in handling long sequences of text as its maximum sequence length is only 512 tokens. This is due to the self-attention operation, which scales quadratically with the sequence length. A possible solution for this issue would be to borrow technical ideas from Longformer (Beltagy et al., 2020),Transformers for Longer Sequences (**B**IG**B**IRD) (Zaheer et al., 2020), and Rabe and Staats (2021) to reduce the quadratic dependency to linear.

Although LUKE was designed for English text, there is an adapted version of this model named m-LUKE (Ri et al., 2021), supporting 24 different languages. One of the proposed ideas is to match Wikipedia hyperlinks that relate to the same article (i.e., *Lisbon* and *Lisboa* are different entity annotations.

However, they share the same entity and Wikipedia article). The detected entity tokens who share the same entities with the input sentence are stacked to the input sequence of the model. This way, entity tokens are expected to provide the model with language-independent features.

3.2 ERICA

A limitation with most previous Transformer-based models is the inability to explicitly model relational facts in the text, which is a major step for textual understanding. Even though some studies have tried to find solutions for this obstacle (Baldini Soares et al., 2019), they only concern individual entities or relationships between pairs of entities within the same sentence. The interactions among more distant entities, whose relations between them are more complex, are completely ignored. To obtain a deeper understanding of entities and their relations in text, Qin et al. (2021) proposed the ERICA model. The authors proposed two new tasks for enabling the model to deal with entities and their relations.

Specifically, the model can be trained using the information of knowledge graph $\mathcal{K}$ to learn all existing entities and relations. For each document d_i belonging to a batch of documents $\mathcal{D}$, all entity pairs (e_{ij}, e_{ik}) are annotated and linked to their possible relation r^i_{jk} present in $\mathcal{K}$. If a pair has no relation annotated in $\mathcal{K}$, it is assigned with a *no_relation* status. Each document has all information regarding the relations between entity pairs in a tuple. Then, the tuples for each document are united to form an overall tuple $\mathcal{T}$. The set of tuples excluding the *no_relation* ones, define the set $\mathcal{T}^+$.

A regular Transformer-based language model encodes each document d_i and outputs a hidden state for each token. To the series of hidden states $\{\mathbf{h}_1, \mathbf{h}_2, ..., \mathbf{h}_{|d_i|}\}$, we apply *mean pooling* over consecutive tokens that mention the j^{th} word of the i^{th} document to obtain local entity representations. Using *mean pooling* to encode entities help the model to handle the problem of representing unknown entities. The index k stands for the number of ocurrences of the entity in the document d_i.

Formally, the k^{th} ocurrence of e_{ij} corresponds to the following equation:

$$\mathbf{m}^k_{e_{ij}} = \text{MeanPool}\left(\mathbf{h}_{n^k_{\text{start}}}, ..., \mathbf{h}_{n^k_{\text{end}}}\right), \tag{25}$$

where n^k_{start}, and n^k_{end} are indexes. To condense information about each unique entity, we set the global entity representation e_{ij} as the arithmetic average of all representations of each ocurrence $m^k_{e_{ij}}$. Afterwards, we concatenate the final representations of two entities e_{ij1} and e_{ij2} as their relation representation $r^i_{j_1 j_2}$. ERICA introduced two new pre-training tasks:

- Entity discrimination task: having into account a given *head* entity and the relation between the pair of entities, the objective of this task is to predict the *tail* entity in a document text. The prediction is then compared to the ground-truth *tail* entity corresponding to the correct relation. This task

motivates ERICA to understand an entity by considering its relation with the remaining entities.
- Relation discrimination task: This task aims to pick two entities and verify whether they semantically relate. In this specific task, the document's targeted entities are distant to comprehend complex relations in real-world scenarios better.

Along with the two novel tasks, the model also uses regular MLM during training for the model not to lose the ability to understand general language (McCloskey and Cohen, 1989).

Formally, the overall learning objective ($\mathcal{L}$) of ERICA is formulated as:

$$\mathcal{L} = \mathcal{L}_{ED} + \mathcal{L}_{RD} + \mathcal{L}_{MLM}. \tag{26}$$

ERICA was benchmarked with BERT and RoBERTa in document-level relation extraction (RE), ET, and QA, outperforming all the baselines. This shows the effectiveness of ERICA in leveraging relations between pairs of entities to learn entity-based representations better. The aforementioned concepts expressed in this section resolves one of the obstacles of LUKE.

However, ERICA shares with LUKE the same limitation regarding handling longer sequences of text at a time. Also, to improve entity representation, ERICA could benefit from the addition or replacement of different methods for entity encoding (i.e., pairs of marker tokens $[S_i], [E_i]$ that denote the limits of each representation of an entity in the document) (Baldini Soares et al., 2019).

4 Thesis Proposal

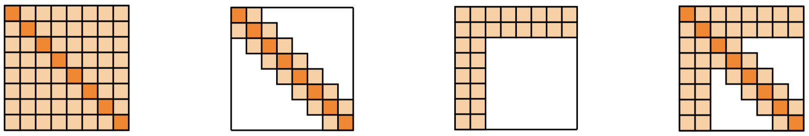
This section proposes the work to be developed during my M.Sc. research. Additionally, I will describe the datasets used in the model and the evaluation metrics.

4.1 Research Statement

This M.Sc. research project has two main objectives. The first one is to develop an entity-aware model capable of receiving longer input text sequences. The second objective is to improve LUKE performance with suitable training objectives. In detail, we seek to prove that being able to effectively use the model's entity improved capabilities over long documents, and the model will afford to have better performances on downstream tasks, such as ET and relation classification.

4.1.1 Proposed Model

As previously mentioned, Transformer-based PLMs are not able to process long input sequences, mainly due to the quadratic time and complexity of self-attention. Recent studies tackle this limitation (Beltagy et al., 2020; Zaheer et al., 2020; Rabe and Staats, 2021), and in our model, we will investigate two



(a) Full n^2 attention (b) Sliding window attention (c) Global attention (d) Combined attention

Figure 7: Proposed configuration of attention patterns.

of the proposals and implement one of the methods. Although the solution presented by Rabe and Staats (2021) is promising, it is not compatible with our architecture implementation.

Longformer (Beltagy et al., 2020) and BigBird (Zaheer et al., 2020) are similar proposals to reduce time and memory complexity. They both propose similar ideas to design and implement a new self-attention pattern, composed of several elements that define the pairs of tokens that will be matched in the attention calculation. The **sliding window attention pattern** (Figure 7b) applies a fixed size window surrounding each token. Several of the aforementioned windowed attention layers are stacked to generate a broad receptive field. Each token attends to $\frac{1}{2}w$ tokens on each side, where w stands for the defined window size. This pattern causes $\mathcal{O}(n \times w)$ complexity, which is linear to the input sequence length n. Up to this point, both techniques propose the same. However, they differ on the method to enhance this pattern. Longformer introduces size dilation gaps d to the window, whereas Bigbird presents the concept of random attention, where queries attend to r random keys. Assuming we have a fixed d and w for l layers, we can have a range of $l \times d \times w$ thousands reachable tokens. In the Longformer implementation, the authors varied the size of the windows across the layers. Particularly, small window sizes are used for lower layers. The window size increases as we move to higher layers, enabling top layers to learn a higher-level representation of the entire sequence while having the lower layers capture local information. Additionally, it also provides a balance between efficiency and performance. Lower layers do not use dilated sliding windows to maximize their capacity to learn the local context. The **global attention pattern** (Figure 7c) is added on pre-defined input locations in order to learn task-specific representations, such as [CLS], [UNK], and [SEP]. In this pattern, the attention operation is symmetric, meaning a token with global attention attends to all tokens across the sequence and vice versa. The number of tokens covered by the global attention is small relative to n. Therefore the complexity of the combined attention patterns is still $\mathcal{O}(n)$. The **combined attention pattern** is shown in Figure 7d. It is possible to observe that it corresponds to the combination of the previous pattern. This method is still effective even when applied only during fine-tuning, which avoids re-training the PLM from scratch.

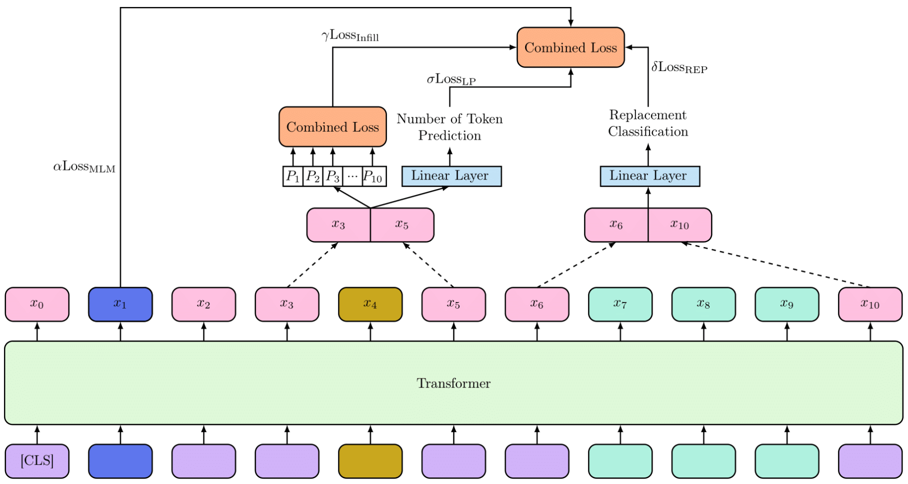

αLoss$_{\text{MLM}}$
γLoss$_{\text{Infill}}$
Combined Loss
σLoss$_{\text{LP}}$
δLoss$_{\text{REP}}$
Combined Loss
Number of Token Prediction
Replacement Classification
P_1 P_2 P_3 ··· P_{10}
Linear Layer
Linear Layer
x_3 x_5
x_6 x_{10}
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}
Transformer
[CLS]

Figure 8: KBIR inspired model architecture for the training phase. The random MASK token is blue, the entity MASK is yellow, and the replaced entity is denoted in cyan.

Our model uses the pre-training tasks presented in LUKE, which enable the model to process documents where complex entities and their relationships should be taken into account. These tasks are especially effective at improving common entity related tasks, such as RE, ET and QA tasks. The first set of tasks was described in ERICA in Section 3.2, particularly the entity discrimination and relation discrimination tasks, which aim to help the model better comprehend entities and their semantics, according to their relations. When we can process long documents, modulating relations becomes even more relevant.

Another similar method to tackle this limitation was introduced by Kulkarni et al. (2021). The authors propose a novel pre-training task Keyphrase Boundary Infillling with Replacement (KBIR) that confers the model the ability to identify keyphrases from a text in an automated way. Even though this pre-training task might not seem to relate to our report subject, the author says the same tools used to learn a rich representation of key phrases can also be used to learn entities. Therefore, we propose to implement a pre-training task inspired by KBIR (Figure 8) to improve our model. We propose to apply a pre-training objective inspired by KBIR which consists of performing a multi-task learning setup, where tasks based on Keyphrase Boundary Infilling (KBI) and Keyphrase Replacement Classification (KRC) are performed concurrently.

In KBI, we replace a complex entity representation with an individual [MASK] token (Lewis et al., 2020), independently of how many tokens the entity is composed of. Afterwards, we use positional embeddings and the masked entity boundary tokens to predict the original tokens of the entity. We define $x_1, ..., x_n$ as the output of LUKE encoder, x_m as the masked original entity y_i, s and e as the indexes of the first and last terms of x_m, and x_{s-1} and x_{e+1} as the boundary tokens of the targeted token. Formally, the predicted entity can be represented as follows:

$$y_i = f(x_{s-1}, x_{e+1}, p_{i-s+1}), \tag{27}$$

where $f()$ represents the activation functions. From the vector representation y_i we compute the cross-entropy loss $\mathcal{L}$ for each token from the original entity x_m. Formally, the aforementioned concepts can be defined as follows:

$$\mathcal{L}_{\text{Infill}}(\theta) = \sum_{i=1} \log p\left(x_i | \boldsymbol{y_i}\right). \tag{28}$$

We train the classifier using a single linear layer trained with cross-entropy loss $\mathcal{L}_{LP}(x_m, z_m)$ along the infilled masked token x_m and the corresponding length of the span class z_m. In addition to the cross-entropy loss calculation, we also predict the expected number of [MASK] entity tokens. The objective is defined as the sum of all cross-entropies of each task, formally:

$$\mathcal{L}_{KBI_{based}}(\theta) = \alpha\mathcal{L}_{\text{MLM}}(\theta) + \gamma\mathcal{L}_{\text{Infill}}(\theta) + \sigma\mathcal{L}_{\text{LP}}(\theta), \tag{29}$$

where α, γ and σ are normalization loss coefficients. With this task, the model learns good representations of entity spans.

The second task, inspired by KRC, gives the model the ability to detect entity spans in input text. We replace entities in the input text with a unique random entity of the same type identified in a knowledge base. Afterwards, the task is modeled as a binary classification task to verify whether the entity is in fact replaced. The input elements of the linear classifier are the boundary tokens x_{s-1} and x_{e+1} of the entity. The goal is to minimize the cross-entropy loss of the classification task $\mathcal{L}_{\text{KRC}}((x_{s-1} + x_{e+1}), y_k)$, where y_k is the label.

At last, our proposed KBIR-based task integrates the objectives of the aforementioned tasks with the purpose that the model learns good entities representation. The final loss is obtained by adding the optimized losses of each performed task, as shown in Equation 30:

$$\mathcal{L}_{\text{KBIR}_{based}}(\theta) = \mathcal{L}_{\text{KBI}_{based}}(\theta) + \delta\mathcal{L}_{\text{KRC}_{based}}(\theta) \tag{30}$$

In our proposal, we will start from the basis of LUKE, availing the more relevant features of this implementation: the entity-aware self-attention mechanism, the dedicated entity embedding mechanism, and the aspect of training with spans of entities. From here, we will introduce relevant features from Longformer and BigBird, and the training tasks we will present, formulated by ERICA and KBIR, will be more relevant to study its efficacy.

4.2 Baselines, Datasets and Evaluation Methodology

This section presents the datasets to be used for testing, some baseline models and the evaluation metrics that will be used to evaluate performance.

4.2.1 Baseline Models

In order to understand whether there is an actual improvement in our implementation, we have to compare it with the models it is based on, LUKE (Yamada et al., 2020) and RoBERTa (Liu et al., 2019).

Infusing Knowledge into Pre-Trained Models with Adapters (K-Adapter), proposed by Wang et al. (2021) is another baseline we will use. It addresses the limitation there is when multiple sources of knowledge are inserted into a PLM like BERT or RoBERTa (i.e., when multiple kinds of knowledge are injected, models hurdle to capture rich factual knowledge (Kirkpatrick et al., 2017)). Therefore, the authors proposed K-Adapter, a framework that supports multiple sources of knowledge into large PLMs. This is achieved with the introduction of Adapters, which are knowledge-specific models annexed outside of a PLM that receive as input the output of hidden states of intermediate layers of the PLM. As adapters are not a module of the PLM, and also have few parameters, their training is independent and memory efficient. K-Adapter is tested on ET and RC to explore the ability of models to learn factual knowledge. On the other hand, to evaluate the impact of the techniques that aim to reduce the time and memory complexity of the self-attention mechanism, we will also use the SciRex (Jain et al., 2020), DyGIE++ (Wadden et al., 2019), and DocTAET (Hou, Jochim, Gleize, Bonin, and Ganguly, Hou et al.) models as our initial baselines. They present current state-of-the-art results in NLP tasks related to long documents.

4.2.2 Datasets and Evaluation Metrics

As general evaluation datasets, we chose OpenEntity (Choi et al., 2018) and FIGER (Ling and Weld, 2012) to test our model in the ET task, and TA-CRED (Zhang et al., 2017) in the RC task. OpenEntity includes 6,000 sentences sampled from web articles. In contrast, FIGER is more fine-grained, covering 112 unique tags. This dataset was developed by exploiting the anchor links in Wikipedia text to label entity segments with appropriate tags automatically. Finally, TA-CRED contains 106,264 sentences, covering 42 relation types between entities. Most existing datasets focus on identifying relationships between a sentence or a paragraph; therefore, it is relevant to use a large-scale dataset at the document level since it requires an understanding of the whole document to annotate entities and their document-level relationships that span beyond sentences.

On the other hand, to test the effectiveness of the implementation of the techniques that enable the model to deal with long documents, we had to choose a suitable dataset, specifically SciREX (Jain et al., 2020). This dataset integrates automatic and human annotations, leveraging existing scientific knowledge resources. SciRex contains 438 documents, an average of 5,737 words, and 22 sections per document. The dataset proposes an evaluation task composed by 3 significant subtasks: (1) identifying individual entities, (2) identifying their document level relationships, and (3) predicting

		True Label	
		Type of target entity	Non Type of target entity
Predicted label	Type of target entity	True Positive (TP)	False Positive (FP)
	Non Type of target entity	False Negative (FN)	True Negative (TN)

Table 1: Confusion matrix for the entity typing task.

their saliency in the document. The dataset is fully annotated with entities, mentions, co-references, and document-level relations.

Regarding document representation, the authors suggested that an input document is represented as a list of sections, and each document is encoded in two steps, section and document-level. Afterwards, pre-trained contextualized token encodings are used over each section separately to get embeddings for tokens in that section. If the section size surpasses 512 tokens, it is broken into 512 token subsections, and each subsection is encoded independently. SciRex requires a global understanding of the entire document to annotate entities and their relations, which may not be possible due to limitations in the implementation. As an alternative, we will also explore the dataset proposed by Jia et al. (2019) that covers single paragraphs composed of multiple sentences.

In order to evaluate the tasks to be performed, we will take into account different evaluation metrics, namely:

1. Accuracy (Acc) is the quotient between correct predictions and the total number of predictions.
2. Precision (P) measures the accuracy of the predictions delivered, dividing the number of correctly predicted observations (i.e., for ET the ground truth and predicted types for a target entity match) with the total number of observations.
3. Recall (R) stands for the ratio of correctly predicted positive observations to all observations in the actual class. We use this metric to select the best model when there is a high cost associated with False Negatives.
4. $\mathrm{F_1}$-score ($\mathrm{F_1}$) is the weighted average of P and R.

The computation of the metrics for binary classification is based on a confusion matrix (Table 1), and is defined according to the following equations:

$$\mathrm{Acc} = \frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}, \tag{31}$$

$$\begin{aligned} \mathrm{P} &= \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}, \\ \mathrm{R} &= \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}, \\ \mathrm{F_1} &= 2 \times \frac{\mathrm{P}\times\mathrm{R}}{\mathrm{P}+\mathrm{R}}. \end{aligned} \tag{32}$$

For evaluating multi-class scenarios, we adopt averaging methods for the $\mathrm{F_1}$ score calculation, resulting in a set of different average scores (i.e., macro,

	OpenEntity (entity typing)			FIGER (entity typing)			TA-CRED (relation classification)		
Method	P	R	micro-F_1	Acc	macro-F_1	micro-F_1	P	R	micro-F_1
LUKE	**79.9**	**76.6**	**78.2**	-	-	-	**70.4**	75.1	**72.7**
K-Adapter	79.3	75.8	77.5	**59.5**	**84.52**	**80.42**	68.9	**75.4**	72.0
RoBERTa	77.6	75.0	76.2	56.31	82.43	77.83	70.2	72.4	71.3

Table 2: Evaluation metrics for a set of baseline methods.

	SciREX (mention identification)			SciREX (end-to-end binary relations)			SciREX (4-ary relation extraction)		
Method	P	R	macro-F_1	P	R	F_1	P	R	F_1
SciREX model	0.707	0.717	**0.712**	**0.065**	**0.411**	**0.096**	**0.531**	0.718	0.611
DyGIE++	0.703	0.676	0.678	-	-	-	-	-	-
DocTAET	-	-	-	-	-	-	0.477	**0.885**	0.619

Table 3: Evaluation of state-of-the-art models on subtasks from SciREX.

weighted, and micro). To evaluate our tasks that contain multiple classes, instead of having multiple per-class F_1 scores, it is preferable to average every score and obtain a single number to describe the overall performance. In order to accomplish this, we can apply three different types of average F_1 score:

- The macro-averaged F_1 score is defined as the arithmetic mean of all the per-class F_1 scores, where all classes are treated equally independently of their support values;
- The weighted-average F_1 score is obtained by taking the mean of all per-class F_1 scores, taking into account the contribution of each class as weighted by the number of examples of that given class.
- The Micro averaging computes a global average of F_1 score by counting the sums of the TP, FN, and FP (Equation 32). Essentially, this computes the proportion of correctly classified observations out of all observations.

The main objective of this evaluation is to excel the performance achieved by LUKE in ET and RC. In Table 2 we have the results of the metrics mentioned above for several baselines in our chosen datasets. In Table 3 we can see the metrics and benchmarks for the subtasks of the task proposed by SciREX. Our goal is to perform the SciREX task in its totality.

5 Summary

This report presented a research project proposal related to leveraging entity-aware representations in NLP tasks. This document presents solutions to the limitations that entity-aware language models face and a summary of the current work. The schedule for this project is presented in Figure 9. Note that the proposed schedule is merely a prediction, and it may be subject to change, depending on the hardware availability and performance expectations.

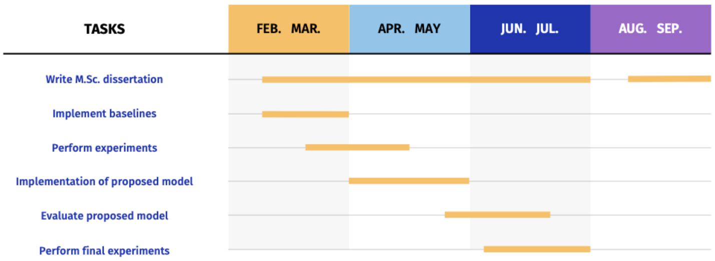


Figure 9: Proposed schedule.

References

Ba, J.L., J.R. Kiros, and G.E. Hinton. 2016. Layer Normalization. *Computing Research Repository (CoRR)* abs/1607.06450 .

Baldini Soares, L., N. FitzGerald, J. Ling, and T. Kwiatkowski 2019. Matching the Blanks: Distributional Similarity for Relation Learning. In *Proceedings of the Association for Computational Linguistics (ACL)*, pp. 2895–2905.

Beltagy, I., M.E. Peters, and A. Cohan. 2020. Longformer: The Long-Document Transformer. *Computing Research Repository (CoRR)* abs/2004.05150 .

Choi, E., O. Levy, Y. Choi, and L. Zettlemoyer 2018. Ultra-Fine Entity Typing. In *Proceedings of the Meeting of the Association for Computational Linguistics (ACL) (Volume 1: Long Papers)*, pp. 87–96.

Devlin, J., M.W. Chang, K. Lee, and K. Toutanova 2019a. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (ACL): Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186.

Devlin, J., M.W. Chang, K. Lee, and K. Toutanova 2019b. Multilingual BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (ACL): Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186.

Ettinger, A. 2020. What BERT Is Not: Lessons from a New Suite of Psycholinguistic Diagnostics for Language Models. *Transactions of the Association for Computational Linguistics (ACL)* 8: 34–48 .

He, K., X. Zhang, S. Ren, and J. Sun 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

He, P., X. Liu, J. Gao, and W. Chen 2021. Deberta: Decoding Enhanced BERT With Disentangled Attention. In *International Conference on Learning Representations.*

Hou, Y., C. Jochim, M. Gleize, F. Bonin, and D. Ganguly. Identification of Tasks, Datasets, Evaluation Metrics, and Numeric Scores for Scientific Leaderboards Construction. In *Proceedings of the Meeting of the Association for Computational Linguistics (ACL)*, pp. 5203–5213.

Jain, S., M. van Zuylen, H. Hajishirzi, and I. Beltagy 2020. SciREX: A Challenge Dataset for Document-Level Information Extraction. In *Proceedings of the Meeting of the Association for Computational Linguistics (ACL)*, pp. 7506–7516.

Jia, R., C. Wong, and H. Poon 2019. Document-Level N-ary Relation Extraction with Multiscale Representation Learning. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (ACL): Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 3693–3704.

Joko, H., F. Hasibi, K. Balog, and A.P. de Vries 2021. *Conversational Entity Linking: Problem Definition and Datasets*, pp. 2390–2397. Association for Computing Machinery.

Joshi, M., D. Chen, Y. Liu, D.S. Weld, L. Zettlemoyer, and O. Levy. 2020. SpanBERT: Improving Pre-training by Representing and Predicting Spans. *Transactions of the Association for Computational Linguistics (ACL)*: 64–77 .

Kingma, D.P. and J. Ba 2015. Adam: A method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR).*

Kirkpatrick, J., R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A.A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell. 2017. Overcoming catastrophic forgetting in neural networks. *In Proceedings of the National Academy of Sciences 114*(13): 3521–3526 .

Kulkarni, M., D. Mahata, R. Arora, and R. Bhowmik. 2021. Learning Rich Representation of Keyphrases from Text. *Computing Research Repository (CoRR)* abs/2112.08547 .

Lan, Z., M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut 2020. Albert: A Lite BERT for Self-supervised Learning of Language Representations. In *International Conference on Learning Representations.*

Lewis, M., Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the Meeting of the Association for Computational Linguistics (ACL)*, pp. 7871–7880.

Ling, X. and D.S. Weld 2012. Fine-Grained Entity Recognition. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, AAAI'12, pp. 94–100. AAAI Press.

Liu, Y., M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. 2019. Roberta: A Robustly optimized BERT pretraining approach, 2019. *arXiv preprint arXiv:1907.11692* 364 .

McCloskey, M. and N.J. Cohen. 1989. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem, Volume 24 of *Psychology of Learning and Motivation*, 109–165. Academic Press.

Modrzejewski, M., M. Exel, B. Buschbeck, T.L. Ha, and A. Waibel 2020. Incorporating External Annotation to improve Named Entity Translation in NMT. In *Proceedings of the Conference of the European Association for Machine Translation*, pp. 45–51. European Association for Machine Translation.

Qian, N. 1999. On the momentum term in gradient descent learning algorithms. *Neural Networks 12*(1): 145–151 .

Qin, Y., Y. Lin, R. Takanobu, Z. Liu, P. Li, H. Ji, M. Huang, M. Sun, and J. Zhou 2021. ERICA: Improving Entity and Relation Understanding for Pre-trained Language Models via Contrastive Learning. In *Proceedings of the Meeting of the Association for Computational Linguistics (ACL) and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 3350–3363.

Rabe, M.N. and C. Staats. 2021. Self-attention Does Not Need O(n^2) Memory. *Computing Research Repository (CoRR)* abs/2112.05682 .

Ri, R., I. Yamada, and Y. Tsuruoka. 2021. mLUKE: The Power of Entity Representations in Multilingual Pretrained Language Models. *Computing Research Repository (CoRR)* abs/2110.08151 .

Ruder, S. 2017. An overview of Gradient Descent optimization algorithms. abs/1609.04747 .

Rumelhart, D.E., G.E. Hinton, and R.J. Williams. 1986. Learning Representations by Back-propagating Errors. *Nature 323*(6088): 533–536 .

Tieleman, T., G. Hinton, et al. 2012. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning 4*(2): 26–31 .

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, and I. Polosukhin 2017. Attention is All You Need. In *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 6000–6010.

Wadden, D., U. Wennberg, Y. Luan, and H. Hajishirzi 2019. Entity, Relation, and Event Extraction with Contextualized Span Representations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 5784–5789. Association for Computational Linguistics (ACL).

Wang, R., D. Tang, N. Duan, Z. Wei, X. Huang, J. Ji, G. Cao, D. Jiang, and M. Zhou 2021. K-Adapter: Infusing Knowledge into Pre-Trained Models with Adapters. In *Findings of the Association for Computational Linguistics (ACL): ACL-IJCNLP 2021*, pp. 1405–1418.

Yamada, I., A. Asai, H. Shindo, H. Takeda, and Y. Matsumoto 2020. LUKE: Deep Contextualized Entity Representations with Entity-aware Self-attention. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6442–6454.

Zaheer, M., G. Guruganesh, K.A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed 2020. Big Bird: Transformers for Longer Sequences. In *Advances in Neural Information Processing Systems*, Volume 33, pp. 17283–17297.

Zhang, W., Y. Feng, F. Meng, D. You, and Q. Liu 2019. Bridging the Gap between Training and Inference for Neural Machine Translation. In *Proceedings of the Meeting of the Association for Computational Linguistics (ACL)*, pp. 4334–4343.

Zhang, Y., V. Zhong, D. Chen, G. Angeli, and C.D. Manning 2017. Position-aware Attention and Supervised Data Improve Slot Filling. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*, pp. 35–45.