Sequence Models

Notes by José A. Espiño P. [1]

Summer Semester 2022–2023

[1]The content in these notes is sourced from what was covered in the MOOG the document is named after. I claim no autorship over any of the contents herein.

Contents

1 Recurrent Neural Networks 2

2 Natural Language Processing and Word Embeddings 7
2.1 Learning Word Embeddings: Word2Vec and GloVe 8

3 Sequence Models and Attention Mechanism 10
3.1 Attention 11

4 Transformer Network 12

1 Recurrent Neural Networks

To represent a word in a sentence, the first thing you have to create is a **vocabulary**, a list of the words that you will use in your representation. We can achieve this by looking through the *n* most common words in our training sets, and then use a one–hot representation to represent each word in our vocabulary. This means that if we have a vocabulary of *n* words, we will have a vector of *n* dimensions, where each dimension represents a word in our vocabulary. The vector will be all zeros except for the dimension that represents the word we want to represent, which will be a one. This is called a **one–hot vector**. There are cases in which we come across words that are not in the vocabulary. When this happens, we can use a special token to represent these words. This token is called **UNK**.
When it comes to processing natural language, standard networks are not very good at it. This is because they do not take into account the order of the words in the sentence. Furthermore, inputs and outputs can be different lengths in different examples. Lastly, regular networks do not share features learned across different positions of text. For instance, if the network can recognise that *Harry* is a name when it is in the first position of a sentence, it will not necessarily be able to classify it as so when it is in the second position.

The solution to this shortcoming is provided by **Recurrent Neural Networks** (RNNs). The key feature of these networks is that they have a **memory** that allows them to take into account the order of the words in the sentence. This memory is represented by a **hidden state** h_t that is passed from one step to the next. Thus, when calculating the activation of a single element in the input sequence, we will not only take into account the input at that position, but also the hidden state from the previous position. At time zero, the hidden state is initialised to zero. Then, at each time step, the hidden state is calculated as follows:

$$h_t = a_t = f(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

where f is an activation function that takes the previous hidden state and the current input and returns the new hidden state.
Furthermore, the output of the network is calculated as follows:

$$y_t = \hat{y}_t = g(W_{ya}a^{<t>} + b_y)$$

where g is an activation function that takes the current hidden state and returns the output. The RNN scans **from left to right** through the input sequence, and the parameters it uses for each step are shared.
One weakness of RNNs is the fact that it only has access to the previous hidden state. This means that it cannot take into account the context of the sentence that comes after the current word. This is where **bidirectional RNNs** come in. These networks have two hidden states, one that scans the input sequence from left to right, and another that scans it from right to left. The two hidden states are then concatenated to form the final hidden state. We will talk more in detail about them later on.
Another weakness of RNNs is that they are not very good at remembering long–term dependencies. This is because the gradient of the loss function with respect to the parameters of the network is multiplied by the same matrix at each time step. This means that the gradient can either explode or vanish as it is propagated through the network. This is called the **vanishing gradient problem**, which we will cover later on.
Forward propagation in an RNN is illustrated in the following image:

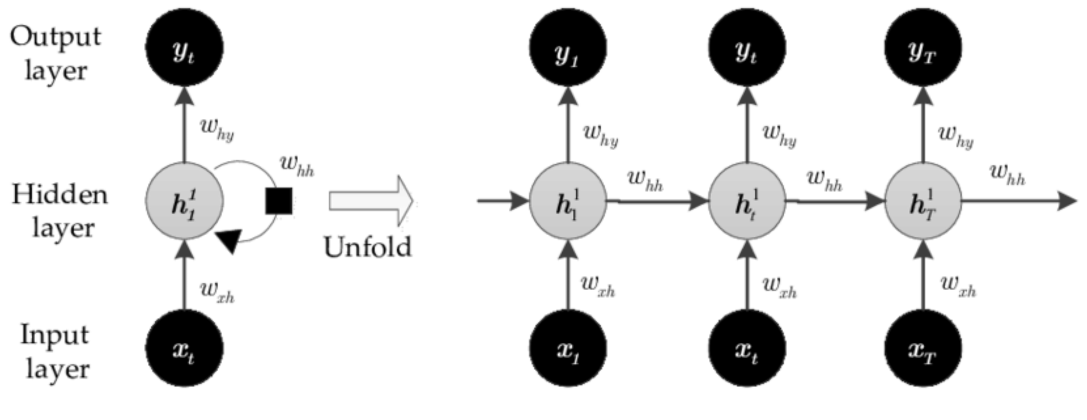

Figure 1: Forward propagation in an RNN, sourced from this site.

When it comes to back propagation, the first thing we need to define is a **loss function**. This function will tell us how good our network is at classifying the input. The loss function we will use is the **cross–entropy loss function**, which is defined as follows:

$$L(\hat{y}, y) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

The equation is a sum because our total loss will be the sum of the losses of each element in the output sequence. You can visualise the back propagation process in the following image:

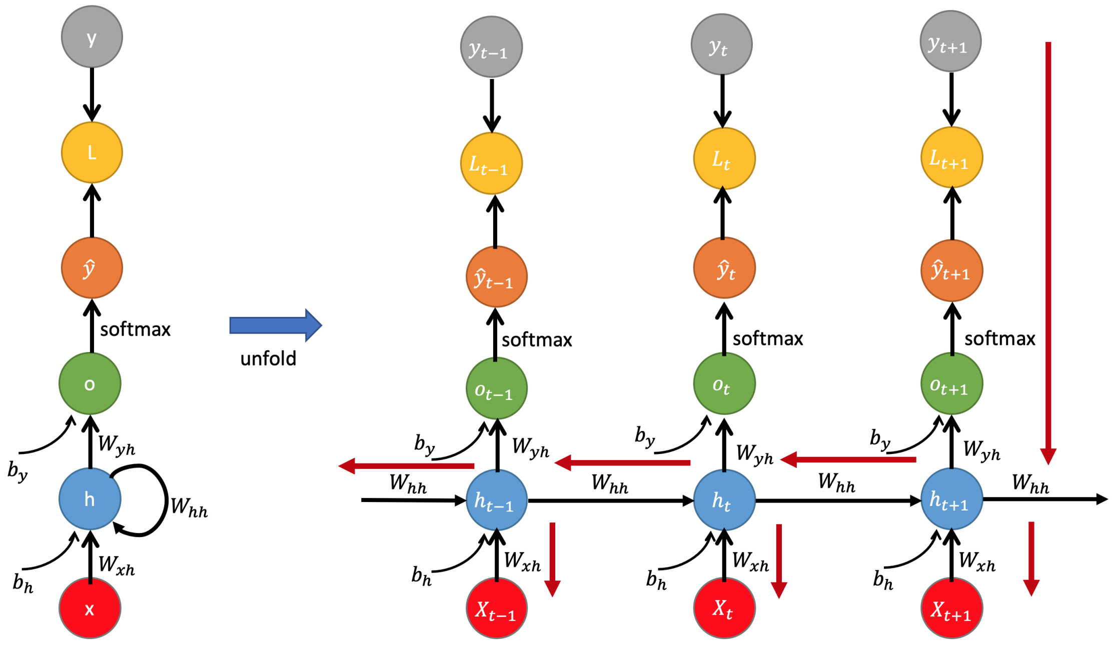

Figure 2: Back propagation in an RNN, sourced from this site.

There are several situations in which the input and output lengths can be different. We can modify the basic RNN architecture to deal with these situations. Some common architectures are:

1. **Many–to–many**: This architecture is used when the input and output are both sequences of the same length. For instance, it can be used to categorise the words in a sentence as per their grammatical function.
   It can also be the case that this kind of architecture is used when the input and output are sequences of different lengths. For instance, it can be used to translate a sentence from one language to another. Usually, they are built as **encoder–decoder pairs**, where the input sequence is first processed by an RNN and encoded into a single vector, and then this vector is used as the initial hidden state of another RNN that will decode it into the output sequence.

2. **Many–to–one**: This architecture is used when the input is a sequence and the output is a single value. Instead of outputting something at each timestep, it will process the entire sequence before outputting a single result. For instance, it can be used to classify a sentence as positive or negative.

3. **One–to–many**: This architecture is used when the input is a single value and the output is a sequence. For instance, it can be used to generate a sentence based on a single word.

4. **One–to–one**: This architecture is used when the input and output are both single values. For instance, it can be used to predict the next word in a sentence.

A language model is a model that can predict the probability of a sequence of words. It can be used to generate text, and it can also be used to evaluate the probability of a sentence.
A language model is built by training an RNN to predict the next word in a sequence, which is done by feeding it a large corpus of text. The RNN is trained to predict the next word in a sequence given the previous words. Before training the RNN, we need to preprocess the text. This is done by first creating a vocabulary of the most common words in the corpus, and then replacing each word in the corpus with its index in the vocabulary — a process known as tokenisation. Furthermore, we need to add a special token to the beginning of each sentence, and another one to the end of each sentence, known as **start of sentence** (SOS) and **end of sentence** (EOS) tokens respectively.
When training the RNN, a common method is **teacher forcing**. This method consists of feeding the RNN the correct word at each time step, instead of feeding it the word it predicted at the previous time step. This is done because the RNN is more likely to learn to predict the next word correctly if it is fed the correct word at each time step.

After you train a sequence model, one of the ways you can get a sense of what it has learnt is to have **sample novel sequences** from it. This is done by feeding it a start of sentence token, and then sampling a word from the probability distribution of the output. Then, you feed the word you sampled back into the network, and repeat the process until you sample an end of sentence token.
RNNs can be built depending on the way you want to represent each element in the sequence. Two of the most common ways are character–level language models and word–level language models. One of the biggest challenges of RNNs is the **vanishing gradient problem**. This is because the gradient of the loss function with respect to the parameters of the network is multiplied by the same matrix at each time step. This means that the gradient can either explode (**exploding gradients**) or vanish as it is propagated through the network. In practice, this means that the RNN will have problems learning long–term dependencies.
A modification of the RNN known as **Gated Recurrent Unit** (GRU) is efficient at capturing long–range connections and alleviate the vanishing gradient problem. As we read an input sequence from left to right, the GRU will have a new variable, c. This is the *memory cell*, which will contain information about the previous hidden states. Considering that at a time t, the value of c will be $c^{<t>}$, the GRU will actually output an activation at t such that $a^{<t>} = c^{<t>}$.
The key of the GRU is it having two gates: the **update gate** and the **reset gate**. The update gate is a vector that determines how much of the previous memory cell is kept. The reset gate is a vector that determines how much of the previous hidden state is kept. The equations for the update gate and the reset gate are as follows:

$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[a^{<t-1>}, x^{<t>}] + b_r)$$

Where σ is the sigmoid function and Γ_u and Γ_r are the update and reset gates respectively.
When it comes to updating the memory cell, the GRU will first calculate a candidate memory cell $\tilde{c}^{<t>}$, which is calculated as follows:

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * a^{<t-1>}, x^{<t>}] + b_c)$$

Once this has been computed, the value of the memory cell is updated as follows:

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1-\Gamma_u) * c^{<t-1>}$$

Finally, the activation at time t is calculated as follows:

$$a^{<t>} = c^{<t>}$$

Another modification of the RNN known as **Long Short–Term Memory** (LSTM) is also efficient at capturing long–range connections and alleviate the vanishing gradient problem. The LSTM is similar to the GRU, but it has three gates instead of two: the **forget gate**, the **update gate** and the **output gate**. The forget gate is a vector that determines how much of the previous memory cell is kept. The update gate is a vector that determines how much of the previous hidden state is kept. The output gate is a vector that determines how much of the current memory cell is outputted. The equations for the forget gate, the update gate and the output gate are as follows:

$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$
$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$
$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$

Where σ is the sigmoid function and Γ_f, Γ_u and Γ_o are the forget, update and output gates respectively.
When it comes to updating the memory cell, the LSTM will first calculate a candidate memory cell $\tilde{c}^{<t>}$, which is calculated as follows:

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$

Once this has been computed, the value of the memory cell is updated as follows:

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$

Finally, the activation at time t is calculated as follows:

$$a^{<t>} = \Gamma_o * \tanh(c^{<t>})$$

The way all of these equations interact in an LSTM is illustrated in the following image:

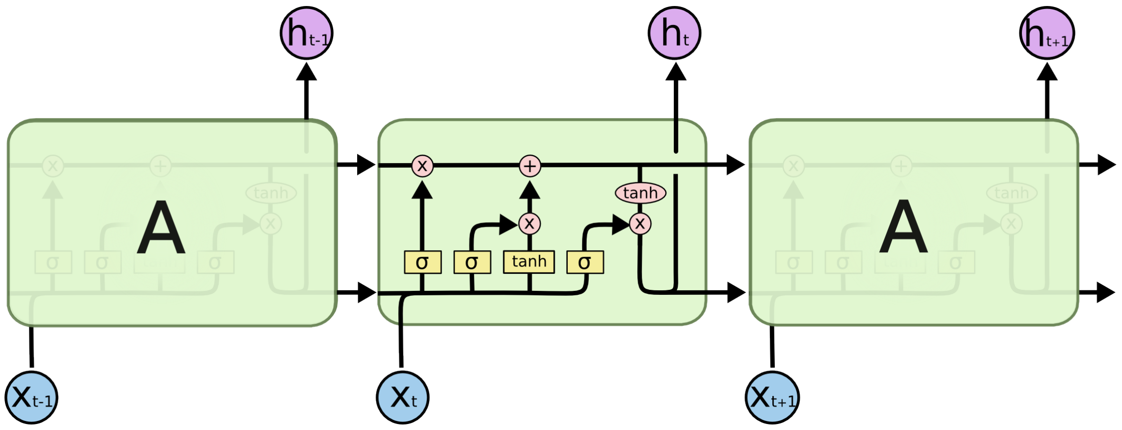

Figure 3: The four interacting layers in an LSTM, sourced from this site.

Another powerful idea is to use **bidirectional RNNs**. These networks have two hidden states, one that scans the input sequence from left to right, and another that scans it from right to left. The two hidden states are then concatenated to form the final hidden state.

Finally, there is the concept of **Deep RNNs**, which are RNNs that have multiple hidden layers. These networks are very powerful, but they are also very computationally expensive.

2 Natural Language Processing and Word Embeddings

Word embeddings are a way of representing words as vectors. These vectors are usually of a much lower dimension than the one–hot vectors we used to represent words in the previous section. The issue with one–hot vector representation is that it treats each word as its own entity, which makes it hard for the network to learn relationships between words. For instance, the one–hot vectors for *king* and *queen* would be completely different, even though they are related. This is because the inner product between any two one–hot vectors is zero.
Word embedding is a featurised representation: we choose a certain set of features (e.g. gender, age, etc.) and then we represent each word as a vector of these features. Using this representation, the embeddings for words that are morphologically similar will also be similar — which may allow the algorithm to learn relationships between words. A common algorithm to visualise the similarity between embeddings on a 2–D plane is the **t–SNE** algorithm.
Word embendding is done through an embedding model that has been trained on billions of words. The most common embedding model is **Word2Vec**. This model is trained by feeding it a large corpus of text, and then it will learn to predict the context of a word given its neighbouring words. Another common embedding model is **GloVe**. This model is trained by feeding it a large corpus of text, and then it will learn to predict the probability of a word appearing given another word.
We use transfer learning with word embeddings by using the embedding model to initialise the embedding layer of the RNN we want to create. This is done because the embedding model has already learnt to represent words in a way that captures their relationships.
An important property of word embeddings is how they can help with *analogy reasoning*. This is done by first computing the distance between two vectors, and then finding the vector that is closest to the result of adding the vector of the word we want to find the vector of and the distance between the two previous vectors. For instance, if we input *"Man is to woman as king is to (blank)"*, the computer will try to find a vector e such that $e_{man} - e_{woman} \approx e_{king} - e$.
The similarity function that is most commonly used is the **cosine similarity**, which is defined as follows:

$$\text{cosine similarity}(u, v) = \frac{u^T \cdot v}{||u||_2 ||v||_2}$$

Usually, we let u be the vector of the word we want to find the vector of, and v be the vector of the word we want to compare it to.

2.1 Learning Word Embeddings: Word2Vec and GloVe

When implementing an algorithm to learn a word embedding, what it ends up learning is a **word embedding matrix** E. This matrix will have a row for each word in the vocabulary, and each row will be the vector representation of the word. Then, if you multiply the one–hot vector of a word by the embedding matrix, you will get the embedding of the word.
Using a neural language model, we can learn a word embedding matrix by training the network to predict the next word in a sequence given the previous words. The embedding matrix will be the weights of the embedding layer of the network. Usually the embedding matrix is initialised randomly, and then it is updated as the network is trained. Additionally, the amount of words before and after the word we want to predict that we feed into the network is called the **context window**, which is a hyperparameter of the network.
Now, let us talk about the **Word2Vec** algorithm. This algorithm's context uses a scheme known as **skip–grams**, which consists of feeding the network a word and then feeding it the words that appear around it in the corpus. The network will then try to predict the words that appear around the word we fed it. Our goal will not be necessarily to predict the words in the vicinity of the target correctly, but rather to learn the embedding matrix. Take a look at this representation of skip–gram:

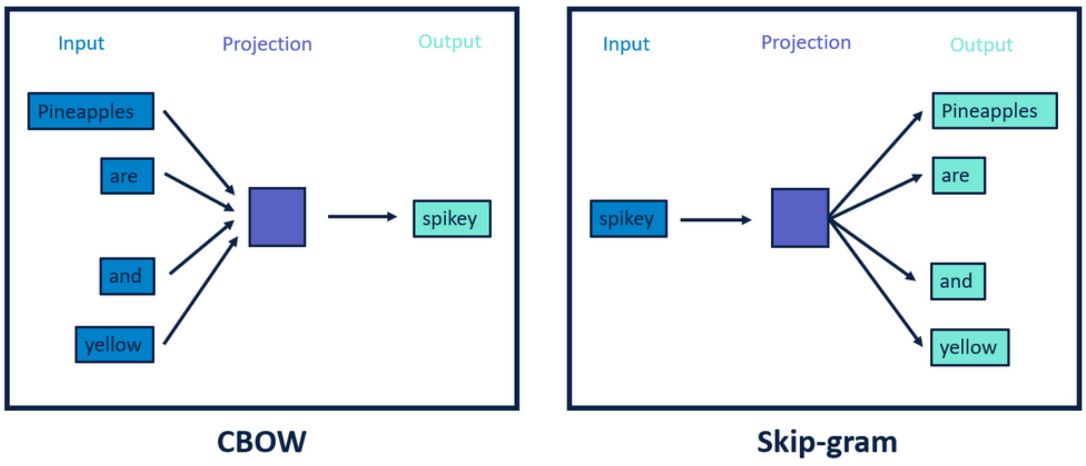

Figure 4: The Word2Vec algorithm

The loss function we will use to train the network is the **softmax loss function**, which is defined as follows:

$$L(\hat{y}, y) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

When we optimise the loss function, we will be updating the embedding matrix.
The main issue with this algorithm is the computational speed: every time we want to evaluate $p(t|c) = \frac{e^{v_t^T e_c}}{\sum_{j=1}^{n} e^{v_j^T e_c}}$, we have to compute the exponential of the inner product of the embedding of the target word and the embedding of the context word. This is computationally expensive, and it is also numerically unstable. We tackle this issue through **Hierarchical**

Softmax, which consists of representing the probability distribution as a binary tree. This way, instead of having to compute the exponential of the inner product of the embedding of the target word and the embedding of the context word, we only have to compute the exponential of the inner product of the embedding of the target word and the embedding of the context word with the nodes of the tree.
A modified learning algorithm that does not use softmax is **Negative Sampling**. This algorithm consists of feeding the model a target word and a word from its context window, and then feeding it a word that does not appear around it in the corpus. The network will then try to predict whether the two word pairs are or not valid context pairs. The amount of negative samples, k is a hyperparameter of the network. We will define a **logistic regression model** to predict whether a pair of words is a valid context pair or not: $P(y=1|c,t)=\sigma(\theta_t^T e_c)$.The loss function we will use to train the network is the **logistic loss function**, which is defined as follows:

$$L(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

On every iteration, we are only going to train $k+1$ binary classifiers, which makes this algorithm much faster than softmax.
Another popular algorithm is **GloVe**, which stands for **Global Vectors**. This algorithm is similar to Word2Vec, but instead of trying to predict the context of a word given its neighbouring words, it tries to predict the probability of a word appearing given another word. The loss function we will use to train the network is the **least squares loss function**, which is defined as follows:

$$L(\hat{y}, y) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

The idea is to essentially minimise the difference between the inner product of the embedding of the target word and the embedding of the context word and the logarithm of the number of times the target word appears in the context of the context word, which is defined as follows:

$$J = \sum_{i=1}^{n}\sum_{j=1}^{n} f(X_{ij})(v_i^T e_j + \log(X_{ij}) + b_i + b_j)^2$$

Here, $f(X_{ij})$ is a weighting function that is used to give more importance to words that appear more frequently. This function is important because it must not overweight common words like *the* and *a*, while not underweight rare words like *onomatopoeia*.
The main difference between Word2Vec and GloVe is that GloVe uses the logarithm of the number of times a word appears in the context of another word, instead of the number of times a word appears in the context of another word.
One of the most basic tasks in NLP is **sentiment classification**, predicting whether the sentiment of a sentence is positive or negative. The simplest way this can be done is by extracting the embedding of each word in the sentence, and then averaging them to obtain the embedding of the sentence. Then, we can feed this embedding into a softmax classifier to predict the sentiment of the sentence. One of the issues of this algorithm is that it ignores word order. Thus, a better way to do it is by using an RNN: we feed the embedding of each word into the RNN, and then we feed the final hidden state of the RNN into a softmax classifier to predict

the sentiment of the sentence. Here, we can consider the RNN as an encoder of sorts.

3 Sequence Models and Attention Mechanism

The core idea behind **Sequence–to–sequence** (Seq2Seq) models is to use two RNNs: one that encodes the input sequence into a single vector, and another that decodes the vector into the output sequence. The first RNN is called the **encoder**, and the second RNN is called the **decoder**. Encoders can even work with images, by removing the last softmax layer from a CNN classification network — this will make the output be a vector representation of the image. This type of model is often used for image captioning, machine translation, speech recognition, etc. Machine translation and conditional language models are essentially different in that the former has two RNNs (encode–decode) while the latter only has one (decode). Machine translation can be seen in terms of conditional probability: model the propbability of the output sequence given the input sequence. Thus, what you are trying to do in this model is to find

$$\arg \max_{y^{<1>},...,y^{<T_y>}} P(y^{<1>},...,y^{<T_y>}|x^{<1>},...,x^{<T_x>})$$

The most common algorithm to maximise this probability is **beam search**. This algorithm consists of keeping track of the B most likely sequences at each time step, and then choosing the B most likely sequences at the next time step from those B sequences. The parameter B (the beam width) determines how many sequences are kept at each time step. The higher the value of B, the more likely it is that the algorithm will find the most likely sequence, but the more computationally expensive it will be.
Beam search is performed by running the sequence through an encoder, and then feeding the output of the encoder into a decoder. The decoder will then output a probability distribution for the next word in the sequence. The B most likely words are then chosen, and then the decoder is run again with each of these words as input ($P(\text{sequence y}|\text{likely word and input sequence})$). Then, the algorithm will see the two word sequences that have been generated and determine which B sequences are the most likely (and then discard the rest). This process is repeated until the end of the sequence is reached.
Length normalisation can help beam search work even more efficiently. This is done by dividing the probability of the sequence by the length of the sequence raised to the power of a hyperparameter α. This way, the algorithm will be more likely to choose shorter sequences. The value of α is usually between 0.5 and 1. Another way to improve beam search is to use **coverage penalties**. This is done by adding a term to the loss function that penalises the algorithm for repeating words. This way, the algorithm will be more likely to choose sequences that have not been chosen before.
When the model outputs a wrong translation, the error can either be found in the RNN or in the beam search algorithm. To determine where the error is, we will input both the correct and the incorrect translation into the model, and then we will compute $P(\text{correct}|x)$ and $P(\text{incorrect}|x)$. If $P(\text{correct}|x) > P(\text{incorrect}|x)$, then the error is in the beam search algorithm. Otherwise, the error is in the RNN.

One of the challenges of machine translation is that there are possibly multiple correct translations for a single sentence. This is why we use **BLEU scores** to evaluate the performance of a machine translation model. This metric is based on the **precision** of the model, which is defined as follows:

$$\text{precision} = \frac{\text{number of words in the machine translation that appear in the reference translation}}{\text{number of words in the machine translation}}$$

The BLEU score is a version of precision in which words can only appear as many times as they appear in the reference translation. This is done to penalise the model for repeating words. Bleu will take unigrams, bigrams, trigrams, and so on to compute the score. This guarantees that the model will be penalised for having incorrect syntax. The BLEU score is defined as follows:

$$\text{BLEU} = \text{BP} \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

Here, p_n is the precision for n-grams, and w_n is the weight for n-grams. The weights are usually set to $\frac{1}{N}$, where N is the number of n-grams. The **brevity penalty** (BP) is defined as follows:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases}$$

Here, c is the length of the candidate translation, and r is the length of the reference translation.

3.1 Attention

The main issue with the Seq2Seq model is that when the sequence gets lengthy, the encoder will have to compress a lot of information into a single vector — which leads to information loss. This is where **attention models** come in. These models allow the decoder to look at the entire input sequence when generating a word. This is achieved by generating **attention weights** α, which determine how much attention the decoder will pay to each word in the input sequence. The attention weights are calculated as follows:

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$$

Here, $e^{<t,t'>}$ is the **alignment model**, which is a function that takes the hidden state of the decoder at time t and the hidden state of the encoder at time t' and returns a value between 0 and 1. The alignment model can be implemented as a neural network.
The context vector is then calculated as follows:

$$c^{<t>} = \sum_{t'=1}^{T_x} \alpha^{<t,t'>} a^{<t'>}$$

4 Transformer Network

So far, all of the models we have covered — RNNs, GRUs, LSTMs — are sequential. This means that they process the input sequence one element at a time, which is practically a bottleneck. This is where **Transformer networks** come in. These networks are non–sequential, which means that they can process the input sequence in parallel. This is achieved by using **attention** to determine how much attention the network should pay to each element in the input sequence.
In order to understand the idea of attention, there are two concepts we need to understand:

1. **Self–attention**: This consists in creating attention–style representations for each of the words in the input sentence. As opposed to word embeddings, this representation tries to make the word vector reflect the meaning of the word within the sentence itself. This means that the representation of a word will be slightly different depending on the sentence it is in.
   Similar to the way we did with RNNs, attention is computed by calculating a **query**, a **key** and a **value** for each word in the input sentence. Then, the attention weights are calculated as follows:

   $$A(q,K,V) = \sum_i \frac{exp(q \cdot k^{<i>})}{\sum_j exp(q \cdot k^{<j>})} v^{<i>}$$

   Here, q is the query, K is the matrix of keys, V is the matrix of values, $k^{<i>}$ is the ith key, $v^{<i>}$ is the ith value, and A is the attention function.
   The query, key, and value of a given word are calculated as follows:

   $$q = W_q x$$

   $$K = W_k x$$

   $$V = W_v x$$

   Here, W_q, W_k and W_v are matrices of weights that are learned during training, and x is the word vector.
   The difference with RNNs is that transformers will compute the alignment model for all of the words in the input sentence at the same time, which means that they can be computed in parallel. Since usually the values of q, K and V are matrices, attention can be computed as:

   $$A(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

2. **Multi–head attention**: This consists in computing attention several times in parallel, and then concatenating the results. This is done because it allows the network to learn different relationships between words. You can think of multi–head attention as just a for–loop that computes attention several times, and then concatenates the results — in practice, this is done in parallel though.

Multi–head attention will have different matrices of weights for the query, key and value for each head, where head h stands for the number of times attention is computed in parallel. This means that the matrices of weights will have the shape $d_{model} \times d_k$, where d_{model} is the dimension of the word vector and d_k is the dimension of the key vector.

Transformers are divided into two blocks, an **encoder** and a **decoder**. The encoder will take the input sequence and output a vector representation of it. The decoder will take the vector representation of the input sequence and output the output sequence. The flow of information in a transformer network is as follows:

1. The input sequence is embedded into a vector representation. This will give us our input.

2. Feed the input into the encoder:

   a) The encoder will compute the values of Q, K, and V to perform self–attention several times in parallel, and then concatenate the results.

   b) Then, the encoder will feed the result into a feed–forward neural network. This helps determine what interesting features are to be found in the sentence.

   In the transformer paper, this encoder block is repeated n times, where n is a hyperparameter of the network.

3. Feed the output of the encoder into the decoder:

   a) First, we will pass `<SOS>` into a multi–head attention block. This will output our first **query**.

   b) Using this first query, we will now pass it alongside the output of the encoder (our first **key** and **value**) into a multi–head attention block.

   c) Then, the values of the multi–head attention block will be passed into a feed–forward neural network. This neural network will determine which is the next word in the sequence.

   Each decoder block will also be repeated n times, where you take the output of one of the blocks and treat it as the input of the other block. Note that all the blocks will have access to the output of the encoder; the only thing that will vary is the starting value that generates the first query.

There are some details that can improve the performance of the transformer network:

- Positional Encoding: Since the transformer network does not have any notion of word order, we need to add a positional encoding to the input sequence. This is done by adding a vector to the word vector that encodes the position of the word in the sentence. This vector is calculated as follows:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

This positional encoding will be directly added to the word vector.

- Add and Norm: This consists in adding a residual connection to the output of each block, and then normalising the result. This is done to prevent the vanishing gradient problem.
- Masked Multi–head Attention: This is used in the decoder to prevent the decoder from looking at future words. This is done by setting the attention weights of the future words to $-\infty$. This way, when the softmax function is applied, the attention weights of the future words will be zero. The point of this is to prevent the decoder from looking at future words, since it is not supposed to know what the future words are.