Απαλλακτική Εργασία στο μάθημα

Advanced Deep Learning

Provide an extensive theoretical description of the RNN-based Machine Translation approach that was adopted throughout the computational project “Word Level Machine Translation” that was presented in class. You should focus on describing the various components of the Encoder – Decoder network architecture both mathematically and graphically.

**High – Level Overview**

At a very high level, the computational project is a Sequence2Sequence problem which uses a model of Encoder – Decoder architecture as its solution.

**Output**

Input

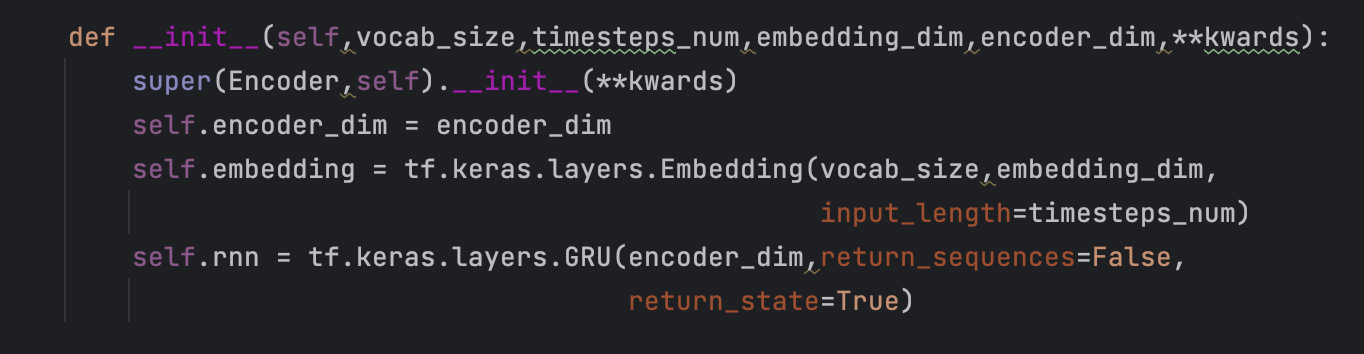
Context Vector

**ENCODER**

**DECODER**

* **Encoder**: The encoder is an RNN which processes each token in the input-sequence. After going through all the tokens, the encoder passes the context vector to the decoder.
* **Context vector**: It is the final internal state of the encoder block, passed onto the decoder block.
* **Decoder**: It reads the context vector and tries to predict the target-sequence token by token.

**Encoder Exploration**

****

As we see, the encoder consists of two compartments:

* The embedding layer
* The GRU Neural Network (Gated Recurrent Unit)

**Explaining the Embedding Layer**

Input

(natural language)

Embedding Layer

Embedding

Vector

0.2

dog

0.8

Since our GRU isn’t able to process natural language, we first want to find a way in order to transform natural language to numbers. Even though we could think of using one-hot vector encoding. This approach is inefficient though. A one-hot encoded vector is sparse (meaning, most indices are zero). Imagine you have 10,000 words in the vocabulary. To one-hot encode each word, you would create a vector where 99.99% of the elements are zero.

Another idea would be to encode each word with a unique number. There are two downsides to this approach, however:

* The integer-encoding is arbitrary (it does not capture any relationship between words).
* An integer-encoding can be challenging for a model to interpret. A linear classifier, for example, learns a single weight for each feature. Because there is no relationship between the similarity of any two words and the similarity of their encodings, this feature-weight combination is not meaningful.

**Word embeddings**

Word embeddings give us a way to create an efficient, dense representation of words. An embedding is a dense vector of floating point values (the length of the vector is a parameter we can specify). Instead of specifying the values for the embedding manually, they are trainable parameters (weights learned by the model during training, in the same way a model learns weights for a dense layer).

Vocab\_size

Embedding\_dim

Cat

Dog

Pet

Queen

King

Man

Woman

Boy

Girl

.

.

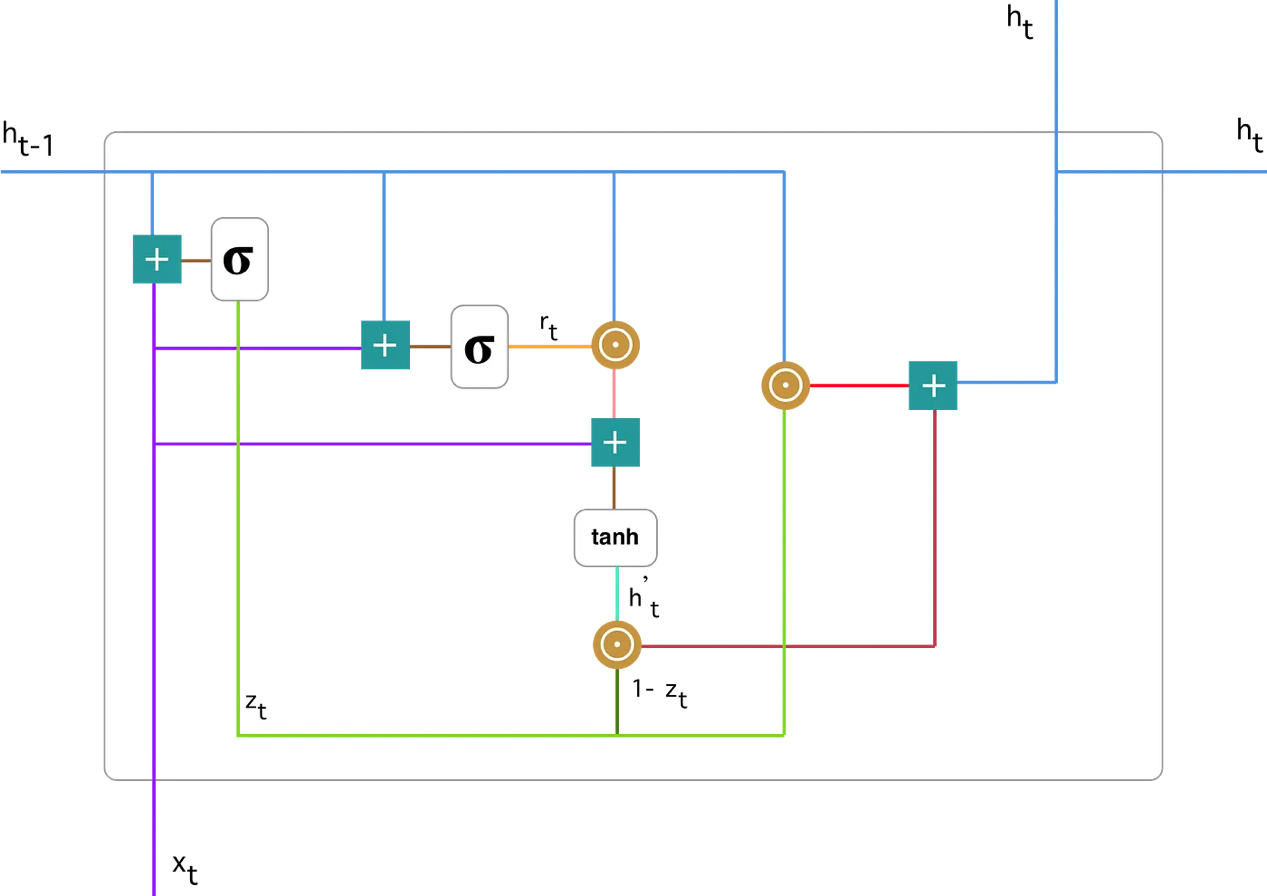
The embedding layer is responsible for creating the trainable matrix (which is instantiated with random values) shown in the above figure. To sum up the functionality of the Embedding Layer, it will first create a trainable matrix of dimensions:

(vocab\_size) x (embedding\_dim)

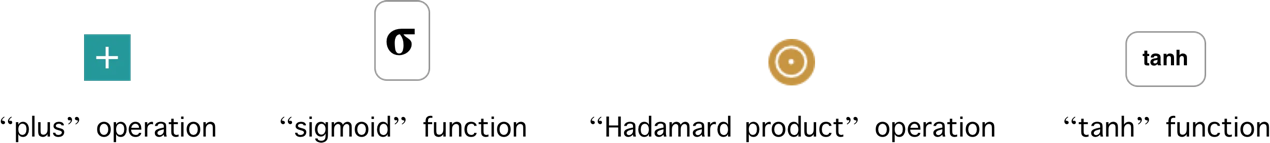
During the forward pass, the embedding vector for each of the tokens in our sentence are going to be obtained from the matrix created before. Because our matrix is trainable during the backward pass (training), the gradients for each of the embeddings are going to be computed and the weights of the matrix will be readjusted accordingly in order to minimize the NN’s loss. The weights are the embeddings themselves.

**GRU NN (Gated Recurrent Unit)**

Introduced by Cho, et al. in 2014, GRU (Gated Recurrent Unit) aims to solve the vanishing gradient problem which comes with a standard recurrent neural network. This is accomplished by implementing an **update gate** and a **reset gate.** Basically, these are two vectors which decide what information should be passed to the output. Below is a detailed view of GRU for a single timestep.

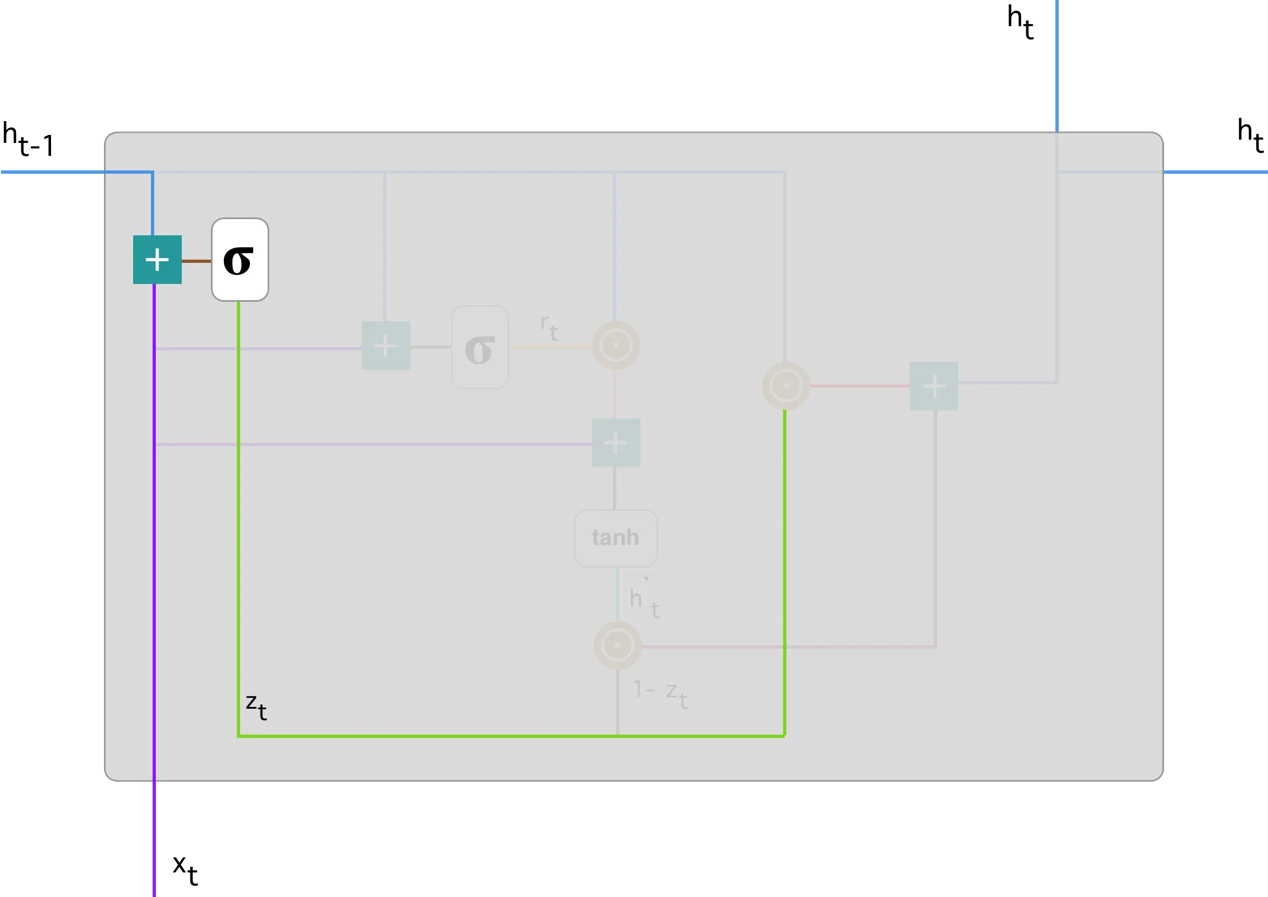


The notations are explained below:



**Update Gate:**

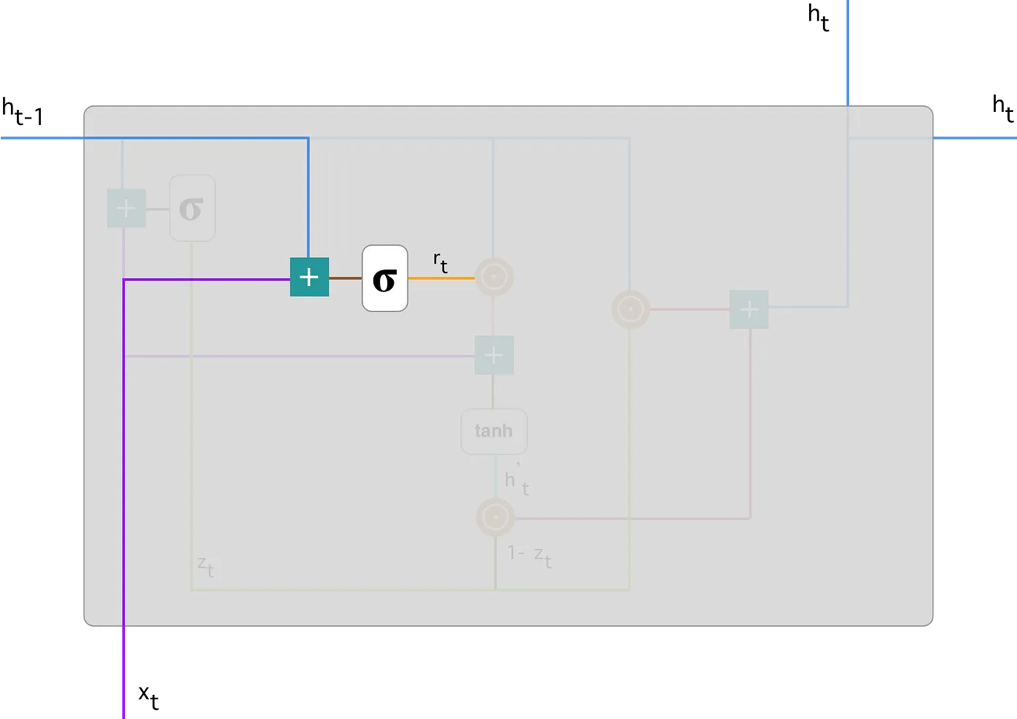
We start by calculating the update gate z\_t for time step t:

When x\_t is plugged into the network unit, it is multiplied by its own weight W(z). The same goes for h\_(t-1) which holds the information for the previous t-1 units and is multiplied by its own weight U(z). Both results are added together and a sigmoid activation function is applied to squash the result between 0 and 1. The update gate helps the model determine how much of the past information needs to be passed along to the future. Following the above schema, we have: 

**Reset Gate:**

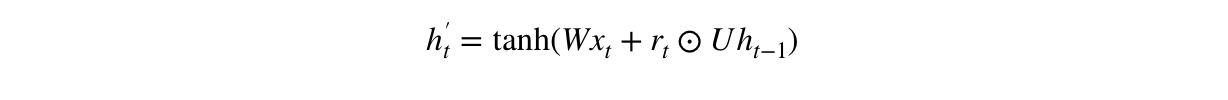
Essentially, this gate is used from the model to decide **how much of the past information to forget.** To calculate it, we use:

This is the same formula with the one used in the update gate but as we can see, weights W\_r and U\_r are different.



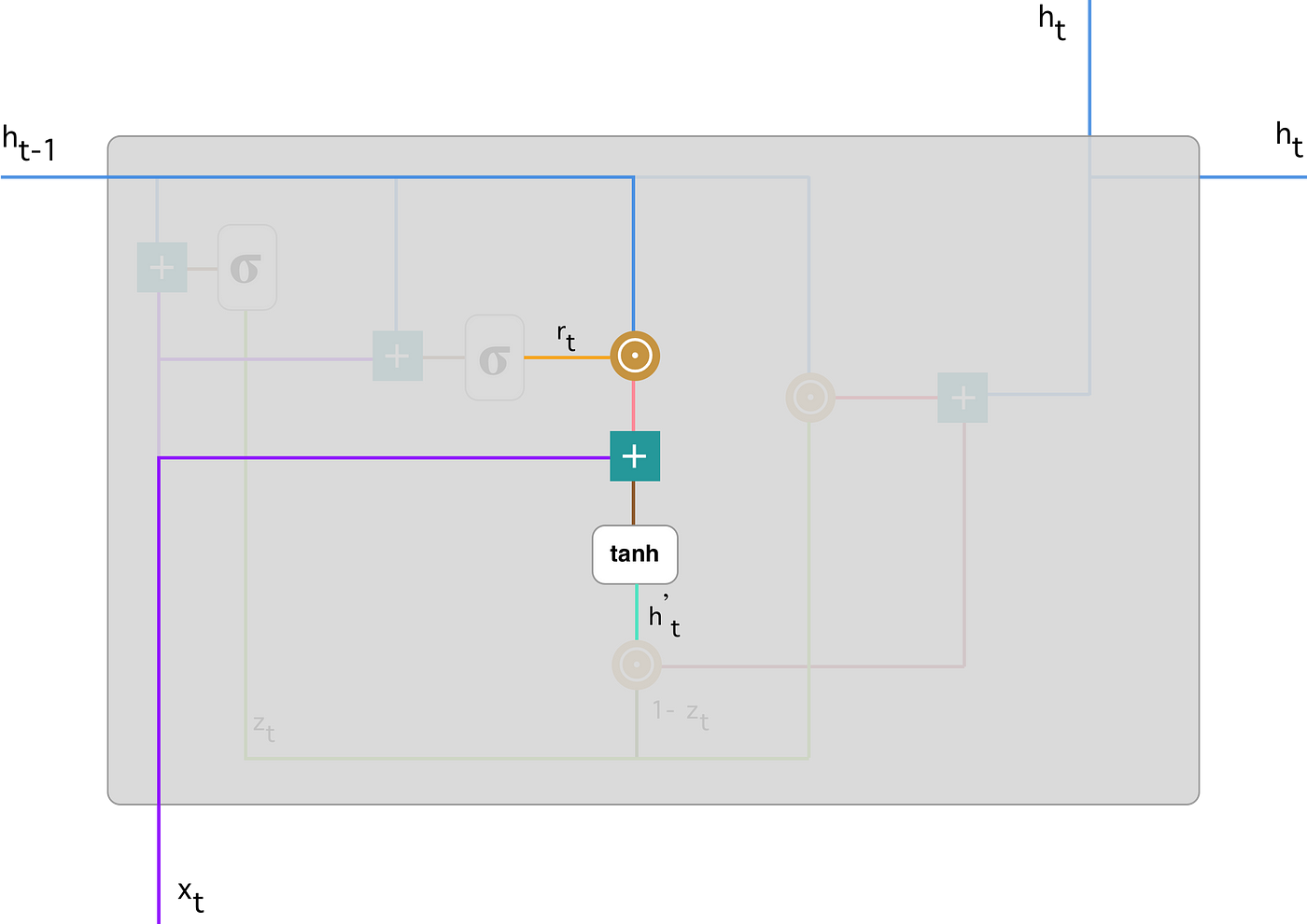
**Current memory content**

Let’s see how exactly the gates will affect the final output. First, we start with the usage of the reset gate. We introduce a new memory content which will use the reset gate to store the relevant information from the past. It is calculated as follows:



1. Multiply the input x\_t with a weight W and h\_(t-1) with a weight U.
2. Calculate the Hadamard (element-wise) product between the reset gate r\_t and Uh\_(t-1). That will determine what to remove from the previous time steps.
3. Sum up the results of step 1 and 2.
4. Apply the nonlinear activation function tanh.

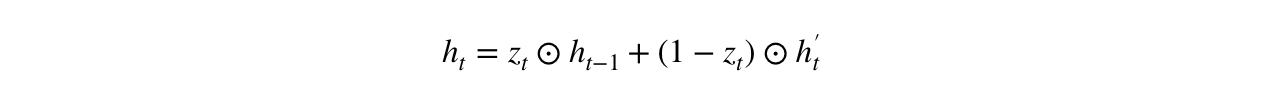
You can clearly see the steps here:



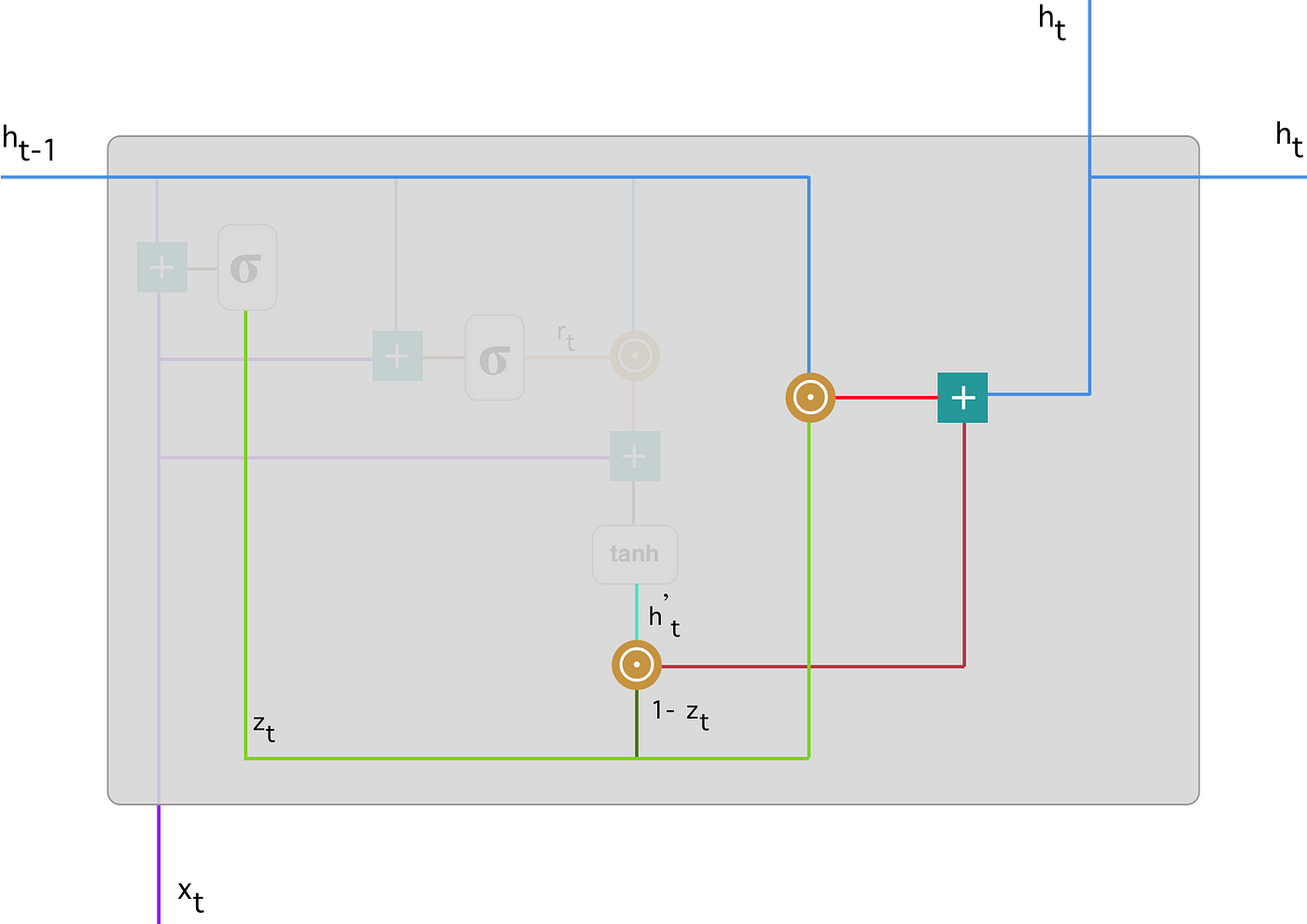
**Final memory at current time step**

As the last step, the network needs to calculate h\_t — vector which holds information for the current unit and passes it down to the network. In order to do that the update gate is needed. It determines what to collect from the current memory content — h’\_t and what from the previous steps —

h\_(t-1). That is done as follows:



1. Apply element-wise multiplication to the update gate z\_t and h\_(t-1).
2. Apply element-wise multiplication to (1-z\_t) and h’\_t.
3. Sum the results from step 1 and 2.

Here is an illustration which emphasizes on the above equation: 

**Decoder Exploration**

The decoder has similar components to the encoder except of having an additional Dense layer whose neurons are equal to vocab\_size. In particular, the decoder consists of:

* Embedding Layer
* GRU Layer
* Dense Layer

**Embedding Layer**

The embedding layer in the decoder is used due to using forced teaching method during training. Force teaching is a training method in which the Decoder’s GRU layer is fed with the ground truth rather than the predicted output from the GRU. This is why we need to have an embedding layer in order to transform the ground truth which is natural language to embedding vectors in order to be used as input for our GRU.

**GRU Layer**

**Predictions**

recontrer

de

ravi

Hidden

state

Embedding Layer

GRU

GRU

GRU

GRU

**Ground Truth**

recontrer

de

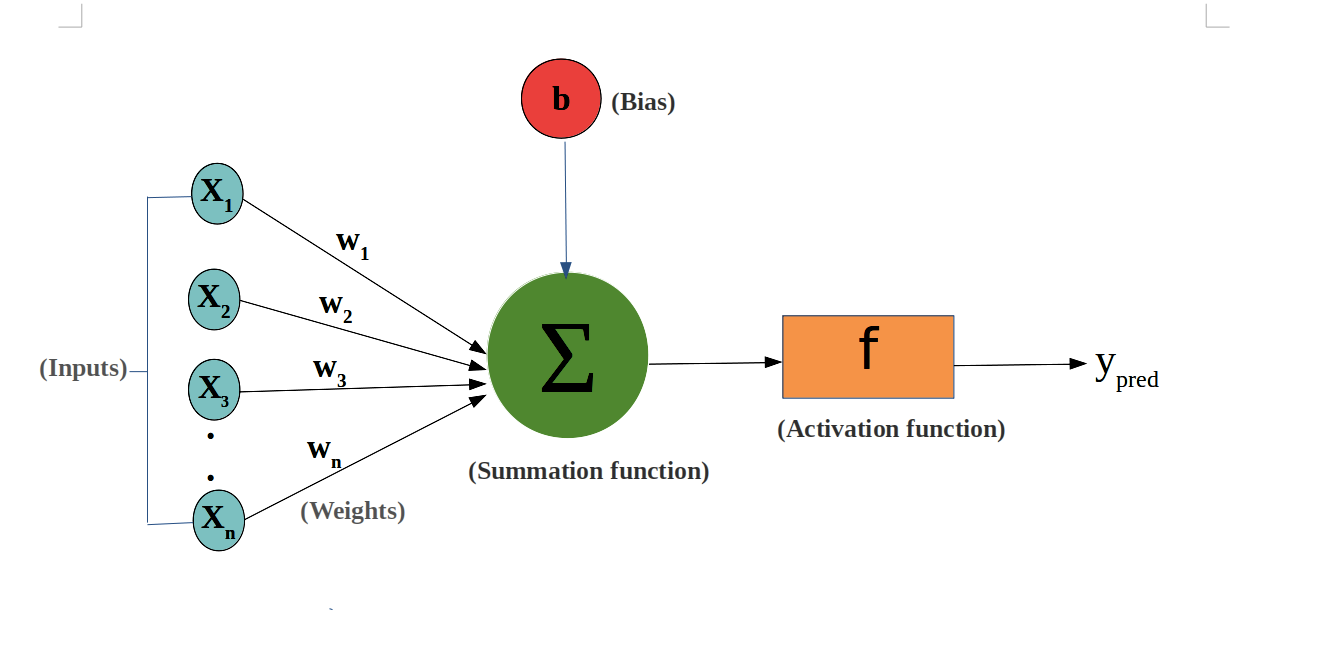
ravi

<SOS>

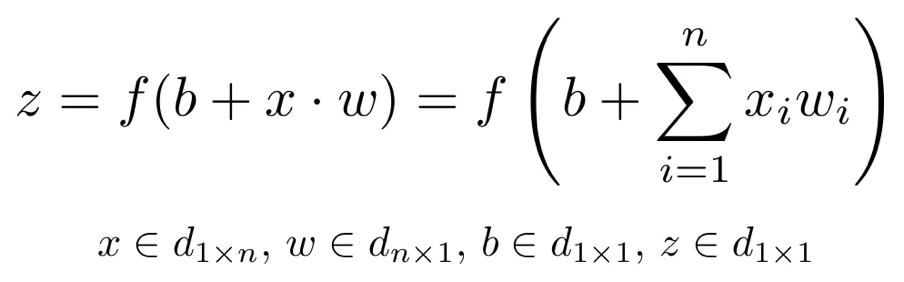
The decoder’s GRU hidden state is initiated with the final hidden state from the encoder’s GRU. The operation of the GRU is fully explained above.

**Dense Layer**

The purpose of the dense layer is to map the outputs of the GRU layer to the vocabulary space. It takes the output sequence from the GRU layer (a vector with size of 1xdecoder\_dim), which captures the contextual information from the input sequence and produces a probability distribution over the vocabulary. Each unit in the dense layer corresponds to a word in the vocabulary, and the output values represent the likelihood of generating each word.



For each of the neurons in the Dense layer which in our case has dimensions of vocab\_size we calculate the output of the neuron as follows:



The neuron with the highest output (which corresponds to a word in the targeted language) predicts the next word in our Seq2Seq model. The weights and biases for each neuron are trained through back propagation in order to minimize the loss. In our case, we have set the loss function to be Sparse Categorical Crossentropy:

**General Overview of the Architecture**

We can look at a general graphically overview of the architecture which was explained above.

Output

GRU

GRU

. . .

Hidden State

GRU

GRU

. . .

Dense Layer

Embedding Layer

Embedding

Vector

Natural Language

Embedding Layer

Ground Truth

(Natural Language)

Embedding

Vector

Provide an extensive theoretical description of the Attention Mechanisms proposed for the problem of Neural Machine Translation presented by the following papers:

1. “**Neural Machine Translation by jointly learning to Align and Translate**” by Bahdanau et al.

The problem which led to developing the attention mechanism in the afore mentioned paper was the following: When having a decoder – encoder architecture similar to the one we had throughout our computational project, the Neural Network will have to compress all the necessary information of the input sentence into a fixed - length vector. This may make it difficult for the NN to cope with long sentences, especially those that are longer than the sentences in the training corpus.

The distinguish feature proposed with this paper is the following: The architecture does not attempt to encode a whole input sequence into a single fixed – length vector. Instead, it encodes the input sequence into a sequence of vectors and chooses a subset of these vectors adaptively while decoding the translation.

**Encoder: Bidirectional RNN for Annotation Sequences**

The proposed scheme uses a BidirectionalRNN which consists of forward and backward RNN’s.

The forward RNN reads the input sequence as it is ordered (from ) and calculates a sequence of *forward hidden states*.

The backward RNN *reads* the sequence *in reverse order* (from ), resulting in a sequence of *backward hidden states*.

We then obtain an annotation for each word by concatenating the forward hidden state and the backward one. i.e. which *contains the summaries* of both the *preceding* and the *following* words. Also, due to the tendency of RNNs to better represent recent inputs, the annotation will be focused on the words around .

**Decoder: General Description**

In the new model architecture, we define each conditional probability as:

Where is an RNN hidden state for time i, computed by:

The probability is conditioned on a distinct context vector for each target word

The context vector depends on a sequence of *annotations ,*which we defined earlier in the BiRNN, to which an encoder maps the input sentence. Each annotation will contain information about the whole input sequence with a strong focus on the parts surrounding the i-th word of the input sequence (based on the tendency of the RNN to better represent recent inputs).

Thus, the context vector is computed as a weighted sum of these annotations :

The weight of each annotation is computed by:

,

Where

Is an alignment model which scores how well the inputs around position j and the output at position i match. We parametrize the alignment model a as a feedforward neural network which in result is jointly trained with all the other components of the proposed system.

We can understand the approach of taking a weighted sum of all the annotations as computing an *expected annotation*, where the expectation is over possible alignments. Let be a probability that the target word is aligned to, or translated from, a source word . Then, the i-th context vector is the expected annotation over all the annotations with probabilities

The probability , or its associated energy , reflects the importance of the annotation with respect to the previous hidden state in deciding the next state and generating . Because of this, the decoder decides parts of the source sentence to pay attention to.

b. “**Effective Approaches to Attention-based Neural Machine Translation**” by Luong et al.

This paper proposes two attention – based models which are classified into two broad categories, *global* and *local.* These two classes differ in terms of whether the “attention” is placed on all source positions (global) or on only a few source positions (local).

Common Ground

The common ground of these two types of models is the fact that at each time step t in the decoding phase, both approaches first take as input the hidden state at the top layer of a stacking LSTM. The goal is then to derive a context vector that captures relevant sourceside information to help predict the current target word . While these models differ in how the context vector is derived, they share the same subsequent steps.

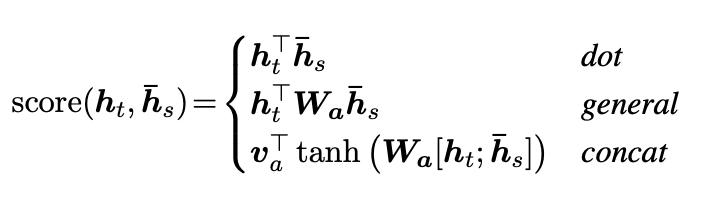
Specifically, given the target hidden state and the source-side context vector , we employ a simple concatenation layer to combine the information from both vectors to produce an attentional hidden state as follows:

The attentional vector is then fed through the softmax layer to produce the predictive distribution formulated as:

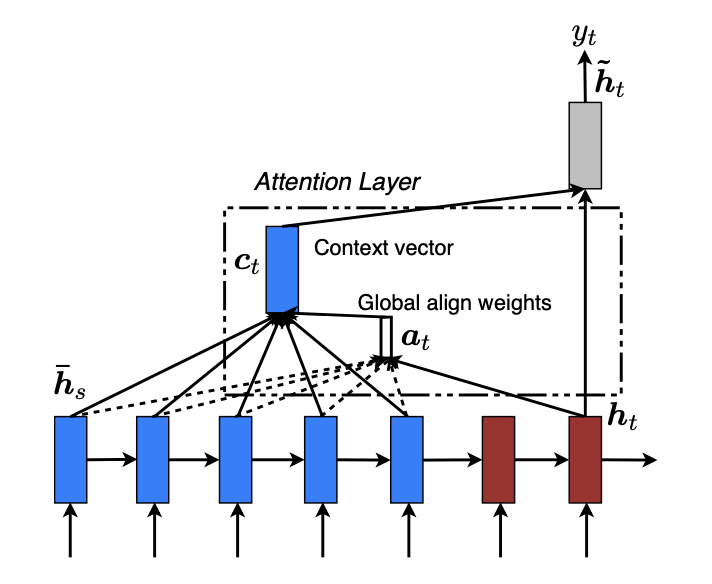
Global Attention

The idea of a global attentional model is to consider all the hidden states of the encoder when deriving the context vector . In this model type, a variable – length alignment vector , whose size equals the number of time steps on the source side, is derived by comparing the current target hidden state with each source hidden state :

Score is referred as a *content-based* function for which we consider three different alternatives:



Given the alignment vector as weights, the context vector is computed as the weighted average over all the source hidden states.



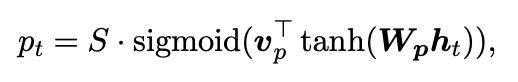
Local Attention

The local attentional mechanism chooses to focus only on a small subset of the source positions per target word.

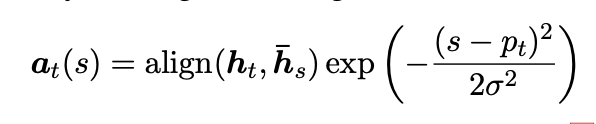
The model first generates an aligned position for each target word at time The context vector is then derived as a weighted average over the set of source hidden states within the window , where D is empirically selected. Unlike the global approach, the local alignment vector is now fixed – dimensional. We now have to consider two variants of the model as below.

*Monotonic* alignment (**local-m**) – we simply set assuming that source and target sequences are roughly monotonically aligned. The alignment vector at is defined as:

*Predictive* alignment (**local-p**) – instead of assuming monotonic alignments, our model predicts an aligned position as follows:



and are the model parameters which will be learned to predict positions. S is the source sentence length. As a result of sigmoid, . To favor alignment points near , we place a Gaussian distribution centered around . Specifically, our alignment weights are now defined as:



We use the same align function as in:

and the standard deviation is empirically set as σ = D.

Note that is a real number; whereas s is an integer within the window centered at .

Difference between the aforementioned approaches and the one presented in class

The approach presented in class initializes the decoder’s hidden state once, using source sentence representation s (a matrix of fixed-length), which in fact is the final hidden state of the encoder. The aforementioned approaches, encode the input sequence into a sequence of vectors and choose a subset of these vectors adaptively while decoding. This later approach is referred to as an attention mechanism.

Elaborate on the major differences between the two attention mechanisms provided by Bahdanau et al. and Luong et al.

In the attention mechanism provided by Luong et al. we simply use hidden states at the top LSTM layers in both the encoder and decoder. On the other hand, use the concatenation of the forward and backward source hidden states in the bi-directional encoder and target hidden states in their non – stacking uni-directional decoder.

Secondly, the computation path is simpler in the attention mechanism from Luong et al. as we go from →→→ and we make a prediction using the following equations:

as for any time t, Bahdanau et al. builds from the previous hidden state →→→, which in turn goes through a deep-output and a maxout layer before making predictions.

Lastly, the paper provided by Luong et al. experiments with more than one alignment function, using dot, general and concat products. In contrast, Bahdanau et al. only use the concat product for the alignment function.