Sequence Models

Nishanth Nair

March 29, 2020

Recurrent Neural Networks (RNNs)

Overview

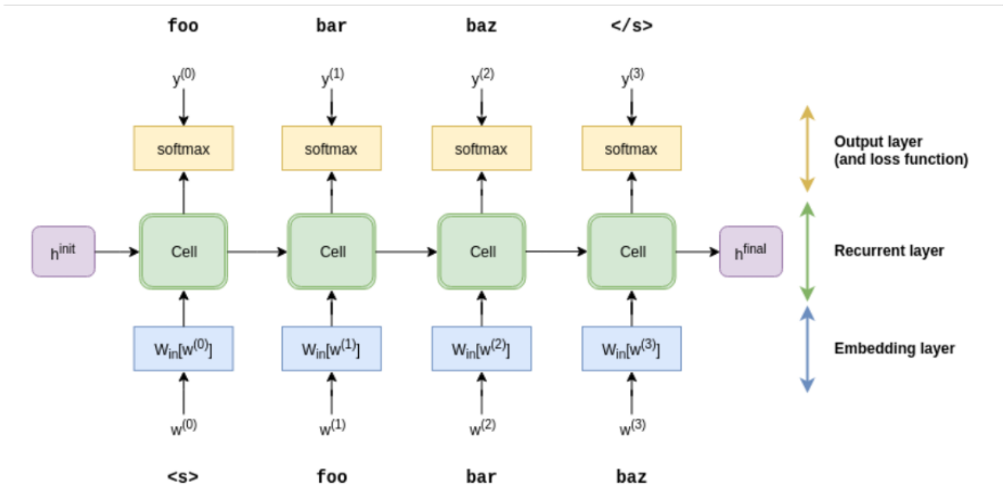


Figure 1: Basic structure of an RNN

- RNNs are a family of neural networks for processing sequential data.
- Parameter sharing makes it possible to extend and apply the model to different forms of data and generalize across them.
- A glaring limitation of regular Neural Networks (and also Convolutional Networks) is that their API is too constrained:
  - they accept a fixed-sized vector as input (e.g. an image) and produce a fixed-sized vector as output (e.g. probabilities of different classes).
  - These models perform this mapping using a fixed amount of computational steps (e.g. the number of layers in the model).
  - The core reason that recurrent nets are more exciting is that they allow us to operate over sequences of vectors:
    - Sequences in the input, the output, or in the most general case both.
- RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being dependent on the previous computations.

- **Advantages**
  - Possibility of processing input of any length
  - Model size not increasing with size of input
  - Computation takes into account historical information
  - Weights are shared across time
- **Drawbacks**
  - Computation being slow
  - Difficulty of accessing information from a long time ago
  - Cannot consider any future input for the current state

Notation and representation

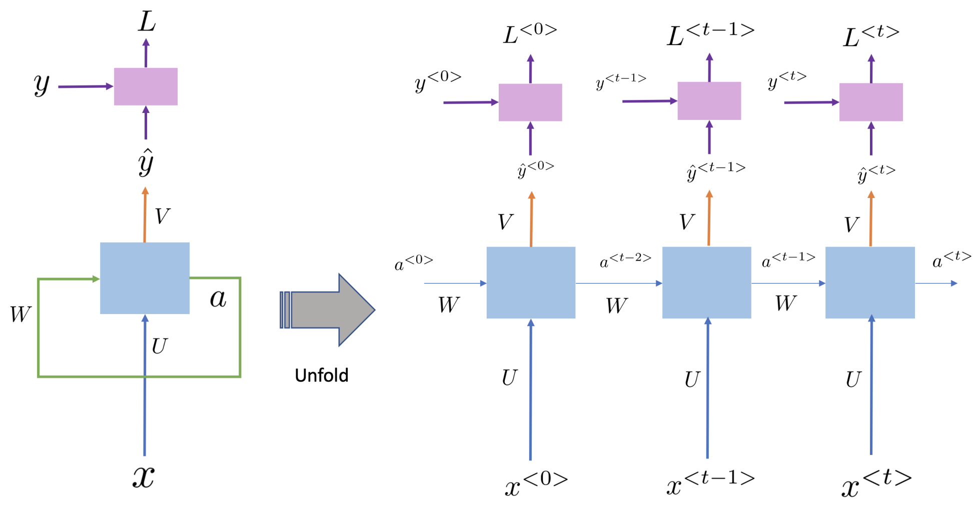


Figure 2: Representation of a RNN

- The parameters U,V,W are shared across time steps.
- $x^{<t>}$ is the input at timestep t
- The activation $a^{<t>}$ is given by:

$$a^{<t>} = f_1(Wa^{<t-1>} + Ux^{<t>} + b_a)$$

- where, function f_1 is any activation func like ReLU or Tanh
- The activation of the previous time step is fed as input to the next times step.
- For thhe initial step, $a^{<0>}$ is typically initialized to zeros.
- The output, $\hat{y}^{<t>}$ is given by:

$$\hat{y}^{<t>} = f_2(Va^{<t>} + b_y)$$

- where, function f_2 is any activation func like ReLU or Tanh or softmax
- The total Loss for a given sequence of inputs and y values is given by:

$$L(\hat{y}, y) = \sum_{t=1}^{T} L(\hat{y}^{<t>}, y^{<t>}) = \sum_{t} L^{<t>}$$

- **Back propogation through time(BPTT)**
  - Back propogation applied to an RNN is called BPTT
  - Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps
    * For example, in order to calculate the gradient at t=4 we would need to backpropagate 3 steps and sum up the gradients.
    * This is called Backpropagation Through Time (BPTT).
  - At timestep T, the derivative of the loss L with respect to weight matrix W is expressed as follows:

$$\frac{\partial L^{<T>}}{\partial W} = \sum_{t=1}^{T} \left. \frac{\partial L^{<T>}}{\partial W} \right|_{(t)}$$

Types of RNNs

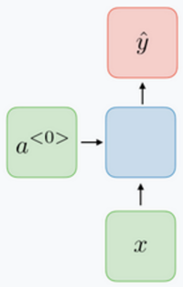
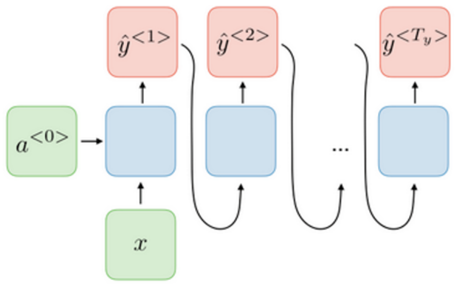
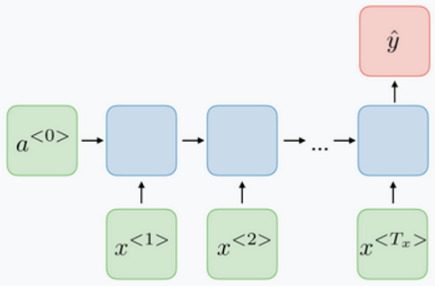
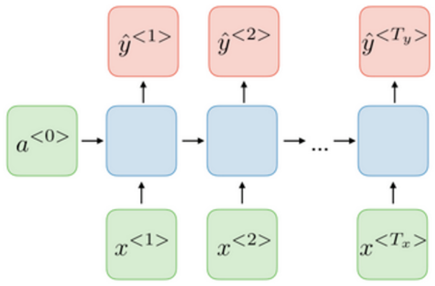
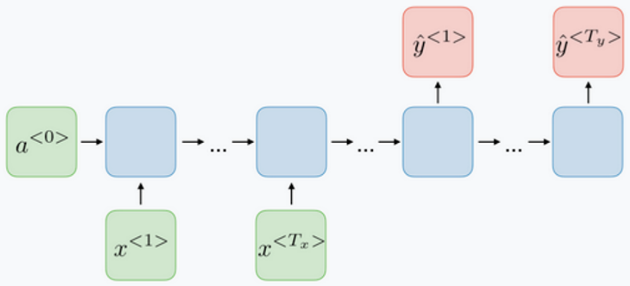
Type of RNN	Illustration	Example
One-to-one $T_x = T_y = 1$		Traditional neural network
One-to-many $T_x = 1, T_y > 1$		Music generation
Many-to-one $T_x > 1, T_y = 1$		Sentiment classification
Many-to-many $T_x = T_y$		Name entity recognition
Many-to-many $T_x \neq T_y$		Machine translation

Figure 3: Types of RNNs

Back-Propogation Through Time (BPTT)

TBD

Handling Long Term dependencies

- Back-propogation propogates the error gradient from the output to the input layer.
  - **Vanishing Gradients**: As the error gradient is propogated to the lower layers, they keep getting small eventually causing the weights close to the input layers to never change.
  - **Exploding Gradients**: Its the opposite of the above, where gradients keep increasing exponentially as it is propogated back causing problems.
  - Why does this happen?
    - Take the example of sigmoid activation.
    - This squeezes the input between 0 and 1.
    - But if you look at values further away from 0, the rate of change in value decreases rapidly and is almost zero.
    - Looking at the derivate below, if we have n hidden layers use an activation like sigmoid, n small gradients are multiplied thus exponentially decreasing the value as we propogare backwards.

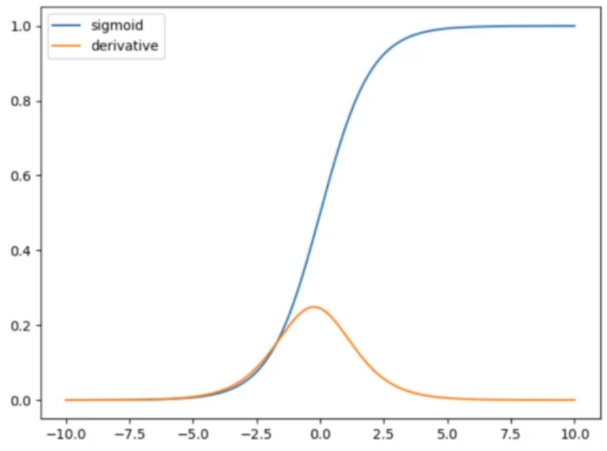


Figure 4: Sigmoid and its derivative

- Some ways to prevent this from happening are as follows:
  - **Activation Function**
    - Use activation functions like ReLU or Leaky ReLU
    - ReLU doesn't saturate positive values.
    - Exponential Linear Unit (ELU) works similarly.
  - **Batch Normalization**
    - Normalize each batch using the batch mean and standard deviation
  - **Weight Initialization**
    - Choosing a different weight initialization (like Xavier initialization) alleviates the problem.
  - **Gradient Clipping**
    - Gradients are cut-off before reaching a pre-determined limit. (example between -1 and 1)

LSTM

Motivation

- In cases where the gap between the relevant information and the place that its needed is small, RNNs can learn to use the past information.
- As that gap grows, RNNs become unable to learn to connect the information.
- As timesteps increase,they “forget” initial inputs as information is lost at each step going through the RNN.
- LSTMs (Long Short-Term Memory) were created to address this.
- LSTMs are explicitly designed to handle the long-term dependency problem.
    - Remembering information for long periods of time is practically their default behavior.
- LSTM units partially solve the vanishing gradient problem, because they allow gradients to also flow unchanged.
- However, LSTM networks can still suffer from the exploding gradient problem

LSTM Cell

- An LSTM cell has the ability to add or remove information to the cell state using carefully regulated structures called **Gates**.
- Gates are a way to optionally let information through.
    - They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.
    - The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.
        - ∗ A value of zero means let nothing through.
        - ∗ A value of one means let everything through.

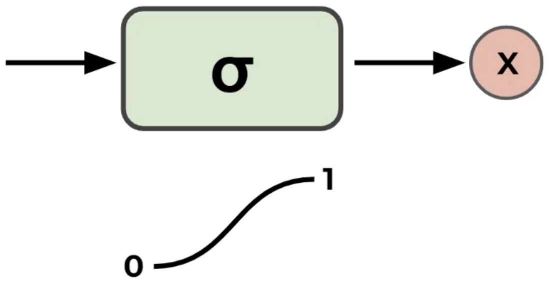


Figure 5: Gate

- The key to LSTMs is the cell state that runs down the entire chain with minor linear interactions.
    - It is very easy for information to just flow along it unchanged.

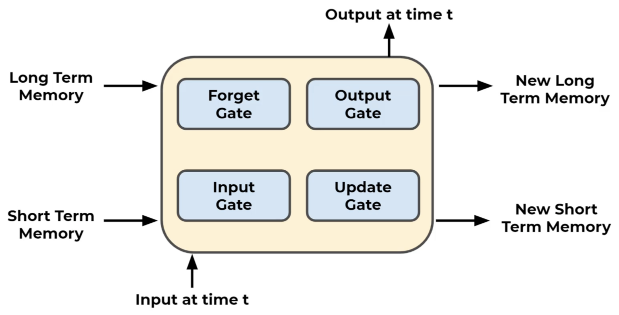


Figure 6: Gates in an LSTM cell

Notation

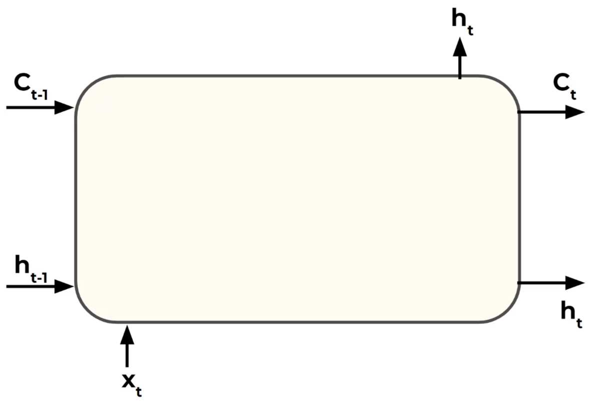


Figure 7: Notation for LSTM

The notation is as follows:

- x_t is the input at time-step t
- h_{t-1} is the hidden state at time-step t-1 (short term input)
- h_t is the hidden state at time-step t (short term output as well as output for cell)
- C_{t-1} is the cell state at time-step t-1 (Long term input)
- C_t is the cell state at time-step t (Long term output)

Mechanics of LSTM cell

Different types of gates are used with a well defined purpose.They are denoted by Γ and expressed as:

$$\Gamma = \sigma(W[h_{t-1}, x_t] + b)$$

Where, W and b are gate specific coefficients.

In an LSTM, the cell works as follows:

- **STEP 1:** Determine what information to keep and what information to discard.
  - This done by the **forget gate (Γ_f)**.
  - looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}.
  - A 1 represents completely keep this while a 0 represents completely get rid of this.

$$\Gamma_f = \sigma(W_f.[h_{t-1}, x_t] + b_f)$$

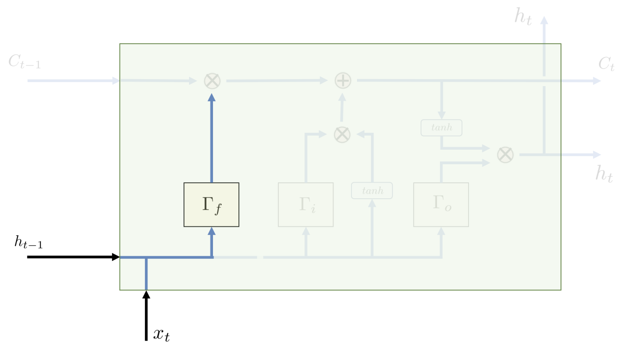


Figure 8: Forget Gate

- **Step 2:** Determine what new information needs to be stored in the cell state.
  - This is done in 2 parts.
  - First, the **Input Gate(Γ_i)** decides which values to update

$$\Gamma_i = \sigma(W_i.[h_{t-1}, x_t] + b_i)$$

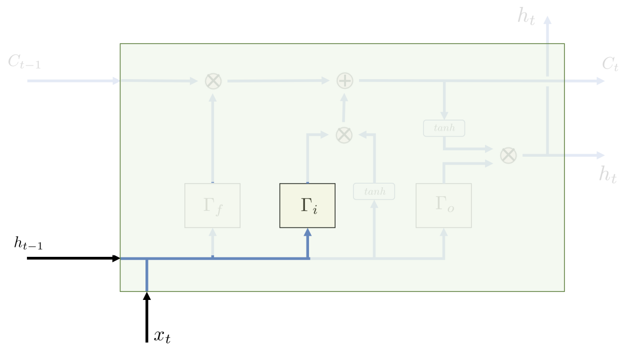


Figure 9: Input Gate

- Next, a tanh layer creates a vector of **candidate values,**$\tilde{C}_t$ that can be added to the cell state.

$$\tilde{C}_t = tanh(W_c.[h_{t-1}, x_t] + b_c)$$

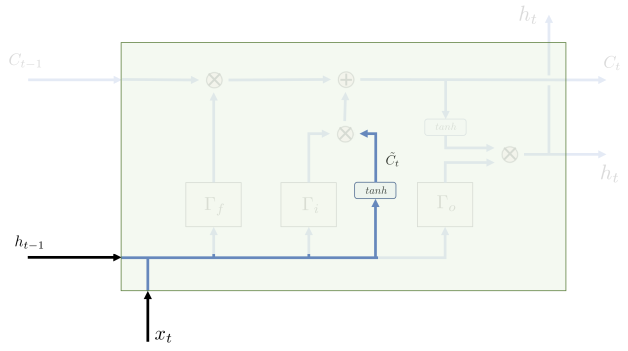


Figure 10: Candidate Values

- **Step 3:** Update the cell state

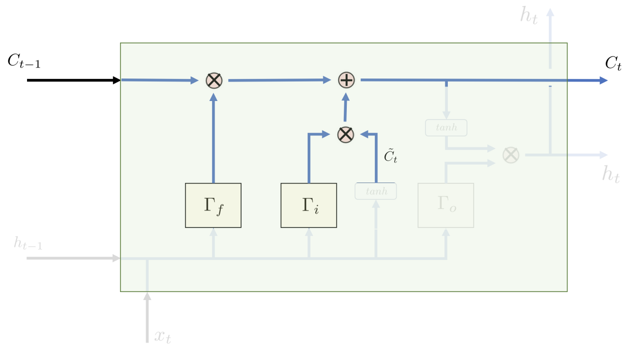


Figure 11: Cell State

- The new cell state is given by:

$$C_t = \Gamma_f \odot C_{t-1} + \Gamma_i \odot \tilde{C}_t$$

- **Step 4**: Determine the output
  - First, the **output Gate(Γ_o)** decides which parts of the cell state to output

$$\Gamma_o = \sigma(W_o.[h_{t-1}, x_t] + b_o)$$

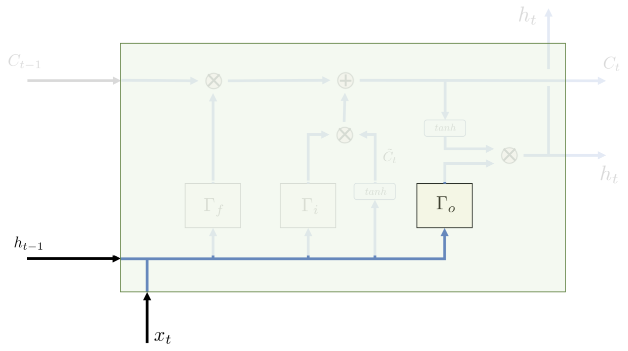


Figure 12: Output Gate

- Next, we put the cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the gate, so that we only output the parts we decided to.

$$h_t = \Gamma_o \odot tanh(C_t)$$

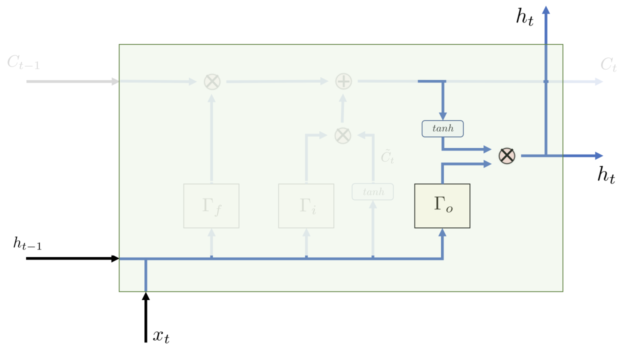


Figure 13: Output

Variants of LSTM

Peephole LSTM

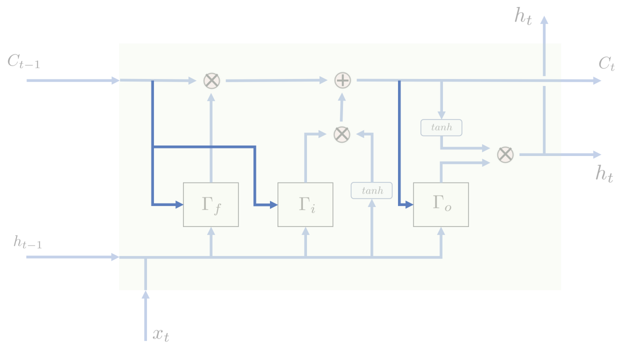


Figure 14: Peephole LSTM

- This is a variation of LSTM that adds "peephole connections"
- This allows the gates to look at the cell state.
- This changes the gates as follows:

$$\Gamma_f = \sigma(W_f.[C_t - 1, h_{t-1}, x_t] + b_f)$$

$$\Gamma_i = \sigma(W_i.[C_t - 1, h_{t-1}, x_t] + b_i)$$

$$\Gamma_o = \sigma(W_o.[C_t, h_{t-1}, x_t] + b_o)$$

Single Update Gate

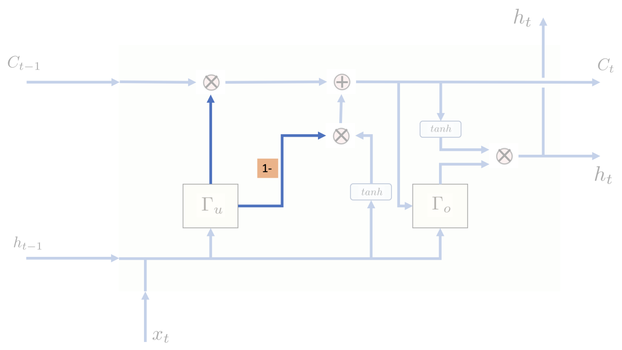


Figure 15: Another variation of LSTM

- In this variation, instead of using a seperate input and forget gate, a single gate called the **update Gate (Γ_u)** is used.
  - We only forget when we are going to input something in its place.
  - We only input new values to the state when we forget something older.

$$\Gamma_u = \sigma(W_u[h_{t-1}, x_t] + b_u)$$

$$C_t = \Gamma_u \odot C_{t-1} + (1 - \Gamma_u) \odot \tilde{C}_t$$

GRU (Gated Recurrent Unit)

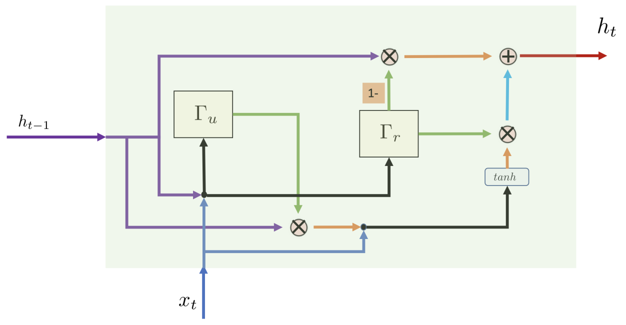


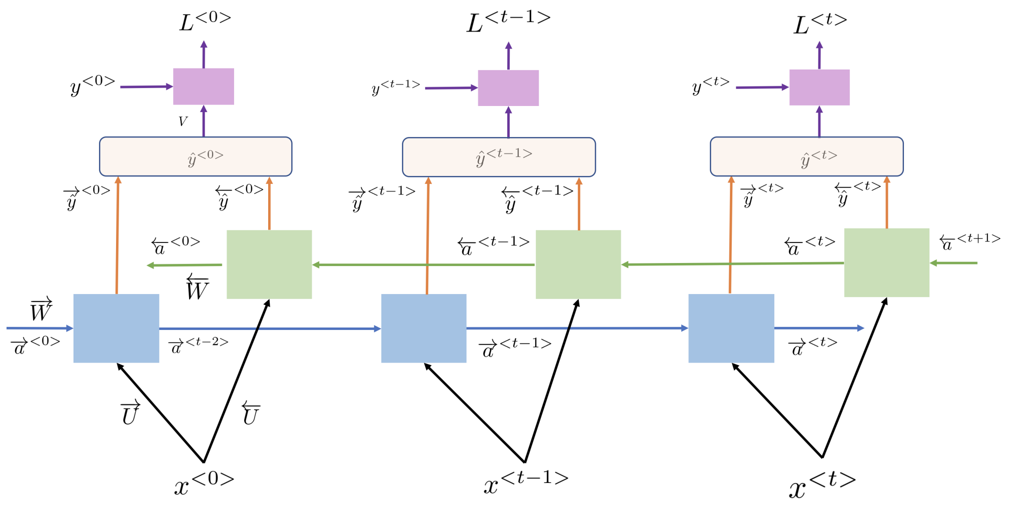
Figure 16: GRU

$$\Gamma_u = \sigma(W_u.[h_{t-1}, x_t] + b_u)$$
$$\Gamma_r = \sigma(W_r.[h_{t-1}, x_t] + b_r)$$
$$\tilde{C}_t = tanh(W_c.[(\Gamma_u \odot h_{t-1}), x_t] + b_c)$$
$$h_t = \Gamma_r \odot \tilde{C}_t + (1 - \Gamma_r) \odot h_{t-1}$$

- A GRU uses 2 gates
- The **reset Gate(Γ_r)** determines how to combine the new input with the previous memory
- The **update gate(Γ_u)** defines how much of the previous memory to keep.
- If we set the reset to all 1s and update gate to all 0s we again arrive at our plain RNN model.
- The basic idea of using a gating mechanism to learn long term dependencies is the same as in a LSTM, but there are a few key differences:
  - A GRU has two gates, an LSTM has three gates.
  - GRUs don't possess and internal memory that is different from the exposed hidden state.
  - They don't have the output gate that is present in LSTMs.
  - The input and forget gates are coupled by an update gate.
  - The reset gate is applied directly to the previous hidden state.
  - Thus, the responsibility of the reset gate in a LSTM is really split up into both r and z.
  - We don't apply a second nonlinearity when computing the output.
- GRUs have fewer parameters and thus may train a bit faster or need less data to generalize.
- On the other hand, if you have enough data, the greater expressive power of LSTMs may lead to better results.

Bidirectional RNNs

- Bidirectional RNNs combine an RNN that moves forward beginning from the start of a sequence, with another RNN that moves backward, beginning from the end of thhe sequence.
- This allows the output units $\hat{y}^{<t>}$ to compute representations that depends on both the past and future but is most senssitive to the input around time step t.



$L^{<0>}$
$L^{<t-1>}$
$L^{<t>}$
$y^{<0>}$
$y^{<t-1>}$
$y^{<t>}$
V
$\hat{y}^{<0>}$
$\hat{y}^{<t-1>}$
$\hat{y}^{<t>}$
$\overrightarrow{y}^{<0>}$
$\overleftarrow{y}^{<0>}$
$\overrightarrow{y}^{<t-1>}$
$\overleftarrow{y}^{<t-1>}$
$\overrightarrow{y}^{<t>}$
$\overleftarrow{y}^{<t>}$
$\overleftarrow{a}^{<0>}$
$\overleftarrow{a}^{<t-1>}$
$\overleftarrow{a}^{<t>}$
$\overleftarrow{a}^{<t+1>}$
$\overleftarrow{W}$
$\overrightarrow{W}$
$\overrightarrow{a}^{<0>}$
$\overrightarrow{a}^{<t-2>}$
$\overrightarrow{a}^{<t-1>}$
$\overrightarrow{a}^{<t>}$
$\overrightarrow{U}$
$\overleftarrow{U}$
$x^{<0>}$
$x^{<t-1>}$
$x^{<t>}$

Figure 17: Bidirectional RNN

- The parameters U,W are shared across time steps.
- $x^{<t>}$ is the input at timestep t and this is fed to both forward and backward pass RNN
- $\overrightarrow{a}^{<t>}$ is the activation of the forward RNN.
- $\overleftarrow{a}^{<t>}$ is the activation of the Reverse RNN.
  - It is given by

$$a^{<t>} = f_1(Wa^{<t-1>} + Ux^{<t>} + b_a)$$

- where, function f_1 is any activation func like ReLU or Tanh
- $\overrightarrow{y}^{<t>}$ is the output of the forward RNN.It is the same as the activation $\overrightarrow{a}^{<t>}$.
- $\overleftarrow{y}^{<t>}$ is the output of the Reverse RNN.It is the same as the activation $\overleftarrow{a}^{<t>}$.
- $\hat{y}^{<t>}$ is the output of the RNN at time step t. It is given by:

$$\hat{y}^{<t>} = f_2(V_y.[\overrightarrow{y}^{<t>}, \overleftarrow{y}^{<t>}] + b_y)$$

- where, function f_2 is any activation func like ReLU or Tanh or softmax

- The total Loss for a given sequence of inputs and y values is given by:

$$L(\hat{y}, y) = \sum_{t=1}^{T} L(\hat{y}^{<t>}, y^{<t>}) = \sum_{t} L^{<t>}$$

Encoder-Decoder Sequence to sequence Architecture

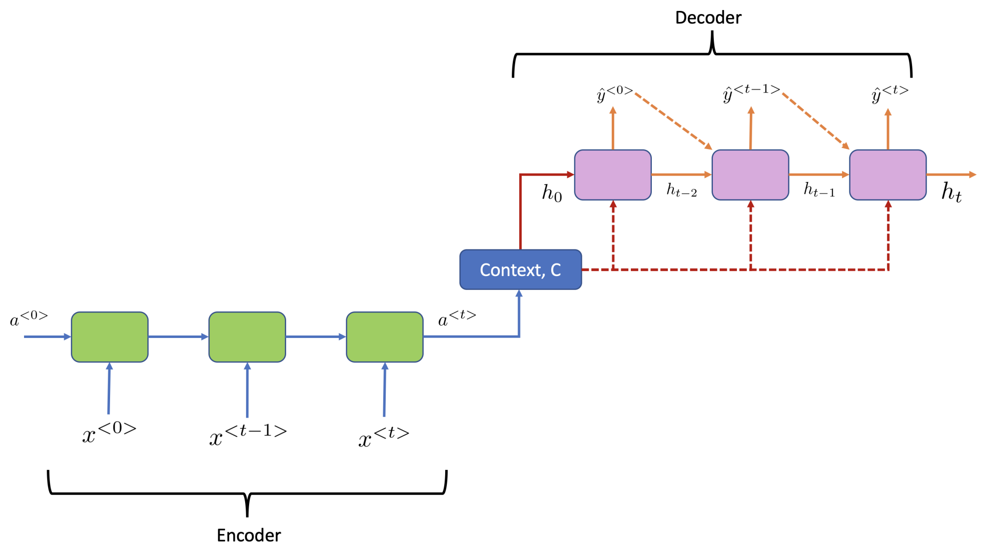


Figure 18: Encoder-Decoder Model

- In this architecture, an RNN can be trained to map an input sequence to an output sequence which is not necessarily the same length as the input.
- The input to the RNN is typically called the "context".
  - The context, C is a vector (or sequence of vectors) representation that summarizes the input sequence X.
- In the encoder-decoder architecture,
  - The **encoder** or reader RNN processes the input sequence and emits the context C (usually as a function of the final hidden state)
  - The **decoder** or writer RNN is conditioned on the context, C to generate the output sequence Y.
    * After a special signal, the decoder starts producing outputs.
    * The decoder keeps generating outputs until a special terminate token is produced.
    * The decoder is often trained to predict the next output given the context vector,C and all previously predicted terms(or words).
    * In other terms,

$$P(\hat{y}) = \prod_{t=1}^{T} P(\hat{y}^{<t>}|\{\hat{y}^{<1>}, ..., \hat{y}^{<t-1>}\}, C)$$

$$P(\hat{y}^{<t>}|\{\hat{y}^{<1>}, ..., \hat{y}^{<t-1>}\}, C) = f(\hat{y}^{<t-1>}, h_t, C)$$

- In sequence-to-sequence architectures,the 2 RNNs are trained jointly to maximize the average of $log(P(y^{<0>}, ..., y^{<n_y>}|x^{<0>}, ..., x^{<n_x>}))$ over all sequences of x,y pairs in the training set.
- There is no constraint that the encoder must have the same size of hidden layers as the decoder.

Attention Mechanism

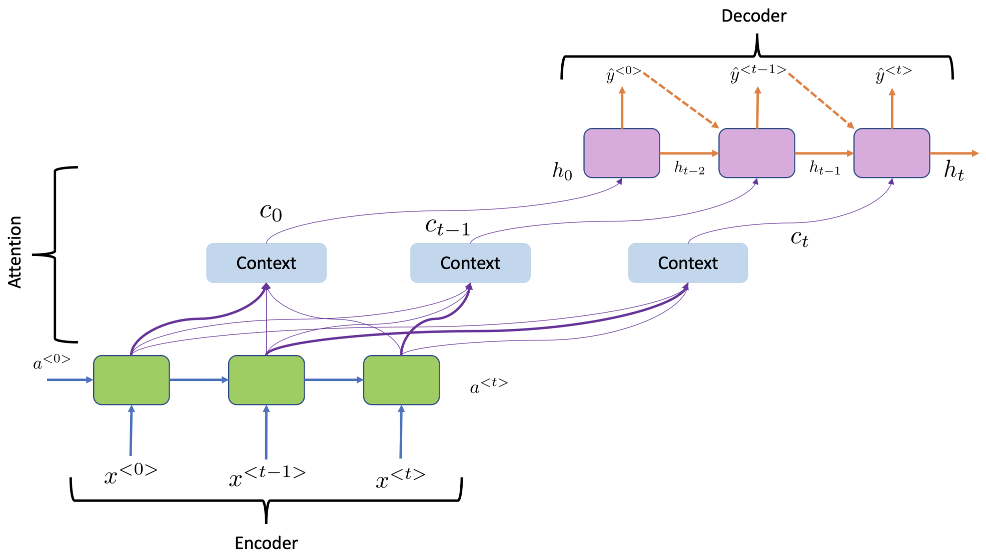


Figure 19: Encoder-decoder with attention Mechanism

- Using a fixed representation to capture all the semantic details of a long input sequence is very difficult.
- The **attention mechanism** is used to focus on specific parts of the input sequence at each time step.
- The general attention based system has 3 parts:
  - A process that reads input data and converts them into a distributed representation of a vector per input feature.
  - A list of feature vectors storing outputs from the reader. This is like a memory containing facts that can be retrieved later, not necessarily in the same order, without having to visit all of them.
  - A process that uses the memory to sequentially perform a task, at each time step paying attention to one or more elements in the memory.

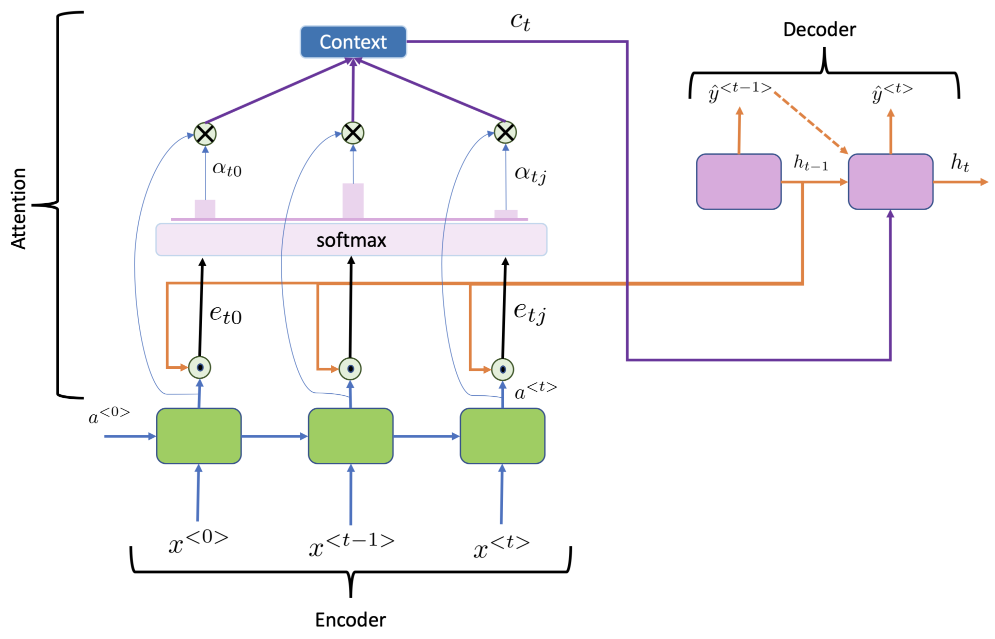


Figure 20: Attention at decoder timestep t

- In this model, unlike the existing encoder-decoder approach, the probability of the output is conditioned on a distinct context vector c_i for each target output $\hat{y}^i$.
- This is expressed as:

$$P(\hat{y}^{<t>}|\{\hat{y}^{<1>}, ..., \hat{y}^{<t-1>}\}, X) = f_1(\hat{y}^{<t-1>}, h_t, c_t)$$

- The context vector c_t depends on a sequence of annotations to which an encoder maps the input sentence.
- An **alignment model** scores how well the inputs around position j and the output position at t match. This is given by:

$$e_{tj} = h_{t-1} \cdot a^{<j>}$$

- The alignment scores are then fed into a softmax to create an attention distribution.
- This is represented by the weight α_{tj}.
  - Since this represents a probability distribution,the α's add up to 1.
  - Higher the value, the more attention the decoder pays to that particular input.

$$\alpha_{tj} = softmax(e_{tj})$$

- The context vector c_t is then computed as follows:

$$c_t = \sum_{j=1}^{T_x} \alpha_{tj}.a^{<j>}$$

- Finally, the hidden state is expressed ass follows:

$$h_t = f_2(h_{t-1}, \hat{y}^{<t-1>}, c_t)$$

- The decoder decides parts of the input to pay attention to.

- By letting the decoder have an attention mechanism, we relieve the encoder from the burden of having to encode all information in the input into a fixed-length vector.With this approach information can be spread throughout the sequence of annotations, which can be selectively retrieved by the decoder accordingly.
- Uses for attention mechanism
  - Voice recognition: allowing one RNN to process the audio and then have another RNN skim over it, focusing on relevant parts as it generates a transcript.
  - Parsing text: where the attention mechaninsm allows the model to glance at words as it generates the parse tree,
  - conversational modeling: where it lets the model focus on previous parts of the conversation as it generates its response.
  - As an interface between a CNN and RNN: This allows the RNN to look at different position of an image every step.
    - Image captioning:
      - First, a CNN processes the image, extracting high-level features.
      - Then an RNN runs, generating a description of the image.
      - As it generates each word in the description, the RNN focuses on the CNNs interpretation of the relevant parts of the image.
- Cost of attention
  - We need to calculate an attention value for each combination of input and output word.
  - If you have a 50 input sequence and generate a 50 output sequence that would be 2500 attention values.
  - But depending on the data, attention mechanisms can become prohibitively expensive.
    - Human attention is something that's supposed to save computational resources.
    - By focusing on one thing, we can neglect many other things.
    - But that's not really what we're doing in the above model.
    - We're essentially looking at everything in detail before deciding what to focus on.
    - Intuitively that's equivalent outputting a translated word, and then going back through all of your internal memory of the text in order to decide which word to produce next. That seems like a waste, and not at all what humans are doing.

Deep Recurrent Networks

TBD

RNN for NLP

TBD

Neural Probablistic Model

Layered RNN

Batch RNN

Sampled or Hierarchical Softmax

Beam Search

BLEU Score