# Sequence Models Course

Sequence models **can learn from temporal / time-series data** (signals that change over time)

Data examples: Speech, Music, Video, Text, Sensor Measurements

Sequence models are trained with supervised learning using labeled training data (x/y)

The training examples x/y can be sequences of data with same or different lengths

When the training examples have different lengths we can use stochastic gradient descent to process them independently. However, if we want to vectorize the propagations for all training examples or for a mini-batch of examples we have to find the longest sequence and pad the smaller ones with zeros in order to create a matrix of sequences with equal length. However, when the training examples have already equal sequence length we can vectorize the propagations in the same way.

Both loss and accuracy can be measured either separately for each output time step or totally for all the time steps after each training iteration / epoch.

Sequence models have many variations / types based on input/output (x/y) types:

**Sequence/Empty Set/Scalar Number/Vector ←To→ Sequence/Scalar Number/Vector**

Some applications:

- Speech recognition: both input (audio clip) and output (text) are sequences
- Music generation: empty set / scalar (as genre) input and sequence output (audio clip)
- Sentiment analysis: sequence input (text) and scalar/vector output (rate level / labels)
- Image Captioning: input is tensor (image) and output is sequence (text)
- Predict DNA parts as protein: both input and output are sequences (DNA sequences)
- Machine translation: both input and output are sequences (text)
- Video activity recognition: input is a sequence (video) and output a probabilities vector
- Named-entity recognition: both input (text) and output (vector of 0/1s) are sequences

**Notation of x and y**

$x^{(i)<t>}$ element at t-th time step of i-th training example input x

$y^{(i)<t>}$ element at t-th time step of i-th training example output y

$T_x^{(i)}$ length of i-th training example input x sequence

$T_y^{(i)}$ length of i-th training example output y sequence

Depending the problem the sequence parts $x^{<t>}$ can be scalars, vectors, matrixes (e.g. the voltage level in case of EGG, 2/3D images in case of grayscale or RGB video, word vectors based on a vocabulary in case of text).

**Why we should not use a feed-forward NN (e.g. MLP) with sequence data?**

To use sequence data with FF NNs we need to unroll/flatten the sequence to an input vector.

1. **Training examples can have different input / output lengths as sequences** and a FF NN input / output layer cannot operate well with variable-length sequences because it has a fixed size. In some cases we can set the input / output layer size to a maximum size that fits the biggest sequence and pad smaller sequences (e.g. with zeros if it has meaning). However, this technique in most cases does not generate good results.

2. **Large number of trainable weight parameters:** A sequence is unrolled/flatten to a single input vector so that can be feed to the FF input layer. For example, imagine a video as a sequence of 30 images of 200x200x3 resolution. The input vector will be 3.600.000 (30x200x200x3) pixel features. Assuming that the first hidden layer has 128 neurons, the weight matrix of the layer will have 460.800.000 trainable parameters (including the 128 biases). It is **very hard to train networks with so many trainable parameters** and most of the times these weight matrixes cannot be stored in the GPU memory (**memory limitations**).

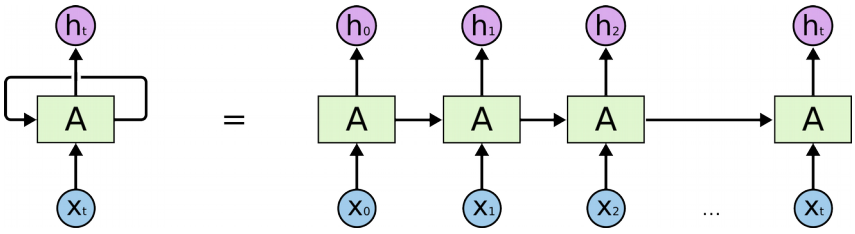
3. **Impossible to share features learned across different positions in data sequences.** For example, if a neuron in a layer has learned a good feature for some input position in a sequence, these features cannot be shared across other positions in the sequence.

(2) and (3) are the same reasons why we use CNN for computer vision instead of MLP models.

**Solution: RNN (Recurrent Neural Networks)**

A recurrent neuron is a neuron that its output is calculated based on its input in the current specific time step and its output from the previous time step, retrieved through a recurrent connection / feedback loop. So, the output of a recurrent neuron in a specific time step it depends not only on the input of the current time step but also on inputs from previous time steps. If we input in a recurrent neuron a sequence of data in a different order we end up with different results. That means that order matters. A recurrent layer is a layer of recurrent neurons and a RNN can be seen as a recurrent layer through time which is used for all sequences of training data.

There are two known diagrams illustrating the basic many-to-many RNN where $T_x = T_y$:




The second diagram (unrolled version) is easier to understand.

The RNN at each time step passes on its activation output to the next time step as extra input.

The activation output in the $0^{th}$ time step is usually a zero vector or a randomly initialized vector.

Most RNN models scan through the training data sequences from left to right.

The trainable parameters ($W_{ax}$ weights connecting input $x^{(i)<t>}$ with hidden layer, $W_{aa}$ weights connecting previous output activation with hidden layer, $W_{ya}$ weights connecting current output activation and output layer) of the RNN are shared across all time steps and training examples. It is actually the same neural network operating through time for many training examples.

Output activation $a^{<t>} = g(W_{aa}\, a^{<t-1>} + W_{ax}\, x^{<t>} + b_a)$

To simplify the calculation of $a^{<t>}$ the $W_{ax}$, $W_{aa}$ weight matrixes can be combined together as $W_a = [W_{ax}, W_{aa}]$ and $x^{(i)<t>}$, $a^{<t-1>}$ input matrixes as $[x^{(i)<t>}, a^{<t-1>}]$ so that:

$a^{<t>} = g(W_a\, [x^{(i)<t>}, a^{<t-1>}] + b_a)$

The $a^{<t-1>}$ is the previous hidden state containing information from the past.

**RNN drawbacks**

1. The classic RNNs (unidirectional RNNs) use at each time step as extra input only computed information from previous time steps. BDRNN (Bi-directional RNN) models at any time step use information both from previous and after time steps.

2. When an RNN is used with data sequences that have large length, the problems of vanishing and exploding gradients appear while training. An RNN is unrolled and is so deep as the length of the sequence that is used. So, when we use lengthy data sequences the RNN seems to be very deep and when we train it the problem of vanishing-exploding gradients appears.

3. Cannot capture and learn information from long time periods.

**BTT (Backpropagation Through Time)**

RNNs are trained using backpropagation and gradient descent. Depending on the RNN architecture and ground-truth labels the appropriate cost function needs to be used to calculate the error. We use backpropagation to propagate back the error gradients and calculate for each recurrent neuron and time step the delta gradient. Using the delta gradients we can update the weight parameters of the RNN.

On calculation of gradients we calculate first the upstream (vertically) gradients $da1^{<t>}$ from the loss to the RNN output activations for each time step. Then we continue, by iterating (horizontally) from last time step to the first one and calculating using $da^{<t>} = da1^{<t>} + da2^{<t+1>}$ (the vertical and horizontal gradient) the $dx^{<t>}$, $da2^{<t-1>}$, $dW_a$, $dW_x$, $db_a$. Also the gradients for the trainable parameters are summed across multiple time steps.

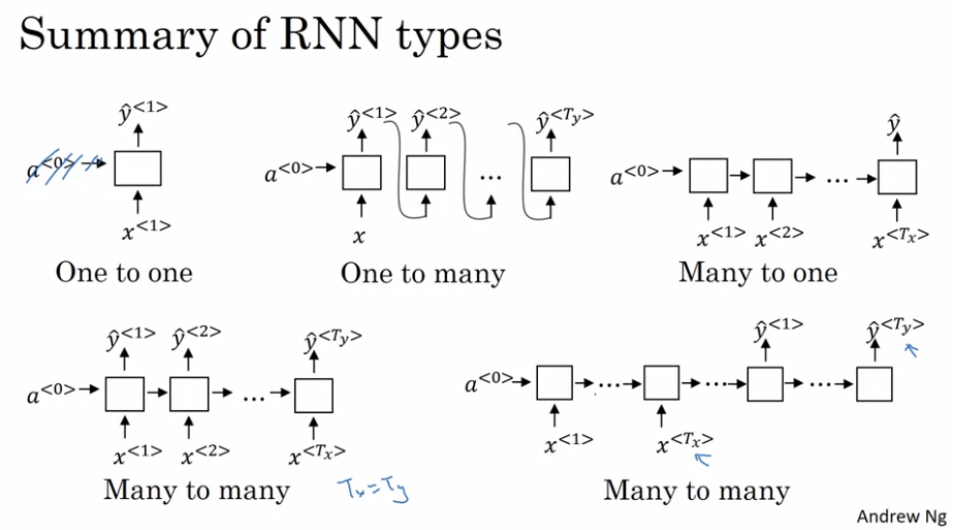
**Types of RNNs**

Classic FF NNs are one-to-one (no sequence in input or output)

RNNs are richer as architectures:

- one-to-many (sequence only in output, e.g. music generation)

- many-to-one (sequence only in input, e.g. sentiment analysis)
- many-to-many $T_x = T_y$ (input and output are sequences, e.g. named-entity recognition)
- many-to-many $T_x <> T_y$ (input and output are sequences, e.g. machine translation)




In the many-to-many ($T_x <> T_y$) architecture we have an "encoder" which digests the total input sequence and generates an output vector which is received by the "decoder" which next decodes it to the output sequence.

**Language Modeling**

Language modeling is useful in many NLP systems (e.g. speech recognition, machine translation). A language model can be built **using an RNN as a many-to-many architecture**. The RNN is trained on text sequences extracted from a very large training corpus of text and outputs how likely (probability) an input text sequence is. In other words, the language model models the chance of text sequences. The language model is trained to learn the next word from left to right.

**Vocabulary:** The training corpus of text needs to be tokenized in order to create a token (words, numbers, punctuation, etc) vocabulary. Then, each token is encoded as a one-hot vector based on the vocabulary.

**Model input / output:** The language model outputs the probability of an input sequence text as $P(y^{<1>}, y^{<2>}, \ldots, y^{<Ty>})$ where $y^{<t>}$ is the one-hot vector of t-th sequence token based on the vocabulary generated.

**Unknown words:** The vocabulary is usually a subset (top-N occurring words) of the set of all words. So, possible unknown words that exist in the training and test datasets are modeled as an <UNKNOWN> token. So, all unknown words have the same "UNKNOWN" one-hot encoding vector.

**End of sentence:** Usually, we train our model with text sequences that are sentences or paragraphs. If we want to use only sentences we need to perform segmentation of text to split sentences. However, if we want to use paragraphs we need to encode also all the characters that mark the end of a sentence as an <EOS> token.

**Examples in speech recognition and machine translation:** Many speech recognition systems use a language model to select the most probable sentence. An example: Assuming we have the following two sentences that sound similar: a) "The apple and pair salad", b) "The apple and pear

salad". Which one is correct? A speech recognition system usually uses a language model that outputs a probability of how likely an input sentence is. Then it selects the most probable (the second sentence in our example). Also, many machine translation systems use a language model in order to ignore a translation in case it is not likely (has low probability to exist in the real world).

**Using a language model we can:**

**a) estimate the probability of an input text sequence as:**

$P(y^{<1>}, y^{<2>}, \ldots, y^{<Ty>}) = P(y^{<1>}) * P(y^{<2>} \mid y^{<1>}) * P(y^{<3>} \mid y^{<1>}, y^{<2>}) * \ldots * P(y^{<Ty>} \mid y^{<1>}, ..., y^{<Ty-1>})$

For a specific input sequence text we multiple the appropriate probabilities from softmax outputs:

$P(o^{<1>})$ [first_word_index] * $P(o^{<2>} \mid o^{<1>})$ [second_word_index] * $P(o^{<3>} \mid o^{<1>}, o^{<2>})$ [third_word_index] * … * $P(o^{<Ty>} \mid o^{<1>}, ..., o^{<Ty-1>})$ [last_word_index]

**b) predict the next token (word / character, etc) having some input sequence of text:** We can input the text sequence to the model and get the probabilities softmax output for the next word and either select the N most probable words as candidates or the word with the largest probability.

**c) generate new text by sampling the softmax output probabilities:** We can randomly sample based on the probability distribution of the first time step's softmax output a candidate word as the first word of our sentence. Then we provide the word's one-hot vector in the next time step as input and random sample again from the softmax output the second candidate word. We continue until <EOS> is found or max N words are sampled randomly. Usually, we perform such a task in order to check what the language model has learned.

**Character-level language model:**

1. The vocabulary is small since it is consisted of single-character tokens
2. The text sequences are long
3. The RNN is deep since the text sequences are long
4. Hard to train (vanishing gradients problem, computationally expensive)
5. We don't need to capture the unknown words with the <UNK> token

**Word-level language model:**

1. The vocabulary is large since it is consisted of words
2. The text sequences are short
3. The RNN is not deep since the text sequences are short
4. Easier to train (less prone to vanishing gradients, needs less computations)
5. We need to capture unknown words with the <UNK> token

**Exploding / Vanishing gradients problems**

In general, very deep NNs suffer from the problems of exploding / vanishing gradients when backpropagation is used to propagate back to the earlier layers the error gradients. Either the gradients vanish or explode as they are propagated back. When they explode the weight updates are catastrophic. When they vanish the earlier layers are either updated too slowly or not updated at all. The exploding gradients problem can be solved with the gradient clipping mechanism which cuts

the gradients when they reach an upper threshold. However, the vanishing gradients problem is difficult to solve. An RNN can be very deep when we use long data sequences and the problem of vanishing gradients can appear on backpropagation through time. This has the side effect that the RNN cannot update its earlier layers in order to affect the activations of later layers.

**In other words, we say that RNNs are not capable of capturing long range dependencies.**

For example, the following two text sequences:

a) The **cat**, ……(long sequence)……., **was** full
b) The **cats**, ……(long sequence)……., **were** full

should be more likely by the language model than the following:

a) The **cat**, ……(long sequence)……., **were** full
b) The **cats**, ……(long sequence)……., **was** full

The language model needs to predict the correct next word (**were / was**) depending the earlier word (**cat / cats**) in the sentence. However, RNNs cannot support long range dependencies due to the vanishing gradients problem where earlier parts of the network cannot be updated to affect later parts. To support long range dependencies and avoid vanishing gradients problem we need to use state-of-the-art RNN models.

**State-of-the-art RNN models: GRU and LSTM**

1. They are based on the basic idea of RNN cell

2. Can capture better the long range dependencies in sequences

   NLP example:

   a) The **cat**, ……(long sequence)……., **was** full
   b) The **cats**, ……(long sequence)……., **were** full

3. They avoid better the vanishing gradients problem

4. They persist and manage internal memory

**Gated Recurrent Unit (GRU)**

The GRU cell introduced a memory cell $c^{<t>}$ which at each time step is influenced by $c^{<t-1>}$ (memory cell information from previous time step) as well as $c\_new\_candidate^{<t>}$ (new memory cell information computed in the current time step). However, $c^{<t>}$ can travel forward along many time steps without significantly changes so that can be used in very later time steps as an extra information coming from the past to help capturing long range dependencies.

In classic RNNs at each time step the output is affected mostly from the earlier previous activations. However, in GRUs the activation output of a time step may be affected also from activations from layers very back in time through the $c^{<t-1>}$ which can carry captured information along the way.

**NLP example usage:** For example, in NLP information could be used in $c^{<t>}$ to capture the fact that the subject of a sentence is singular or plural so that later parts of the sentence can use this

information to create a long range dependency and thus give a higher probability to the next singular or plural verb word (e.g. cat…..was, cats…..were).

**GRU Input / Output:** At each time step the GRU retrieves previous memory cell information from $c^{<t-1>}$ as well as the input $x^{<t>}$ from current time step and generates new memory cell information $c^{<t>}$ for the next time step.

**The GRU cell has 3 learnable layers:** The GRU has a trainable layer with $c^{<t-1>}$ and $x^{<t>}$ as inputs that generates a vector of update gates $\Gamma_u^{<t>}$ (values with [0-1] range) which control which and how much information will be retrieved from c_new_candidate$^{<t>}$ and $c^{<t-1>}$ for calculating $c^{<t>}$ for the new time step. Furthermore, the GRU has another trainable layer with $c^{<t-1>}$ and $x^{<t>}$ as inputs that generates a vector of relevance gates $\Gamma_r^{<t>}$ (values with [0-1] range) which control the amount of information is used by $c^{<t-1>}$ to generate c_new_candidate$^{<t>}$. Lastly, the c_new_candidate$^{<t>}$ is generated by a trainable layer which retrieves as input $c^{<t-1>}$, $x^{<t>}$ and $\Gamma_r^{<t>}$. So, while training, the GRU learns which information from $c^{<t-1>}$, c_new_candidate$^{<t>}$ and how much of it to use (forget or remember) to generate $c^{<t>}$ as well as what $c^{<t-1>}$ information and how much of it will be used to generate c_new_candidate$^{<t>}$.

**Compared to LSTM:** Simpler (easier to train and build complex models) but less powerful.

**2 gates used:** Update $\Gamma_u^{<t>}$ and Relevance $\Gamma_r^{<t>}$ gates

**Dimensions:** The $c^{<t-1>}$, $c^{<t>}$, c_new_candidate$^{<t>}$, $\Gamma_u^{<t>}$ and $\Gamma_r^{<t>}$ are vectors with identical dimensions.

**Math of GRU:**

GRU

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

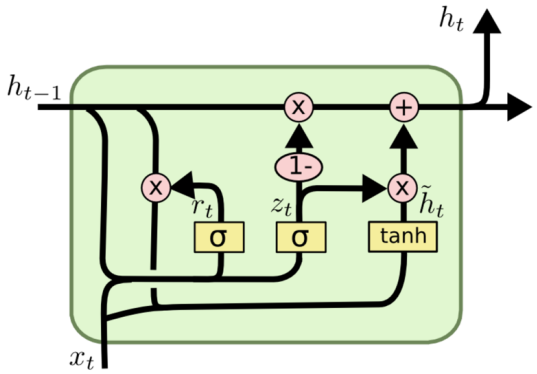
$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

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

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

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

**Cell diagram:**




(b) Gated Recurrent Unit

**Long Short Term Memory (LSTM)**

**Compared to GRU:** More powerful but more complex (harder to train and compose bigger models), default choice in most cases.

**LSTM Input / Output:** At each time step the LSTM retrieves previous memory cell information from $c^{<t-1>}$, previous activation output $a^{<t-1>}$ as well as the input $x^{<t>}$ from current time step and generates new memory cell information $c^{<t>}$ and activation output $a^{<t>}$ for the next time step.

**3 gates used:** Input $\Gamma_i^{<t>}$, forget $\Gamma_f^{<t>}$ and output $\Gamma_o^{<t>}$ gates.

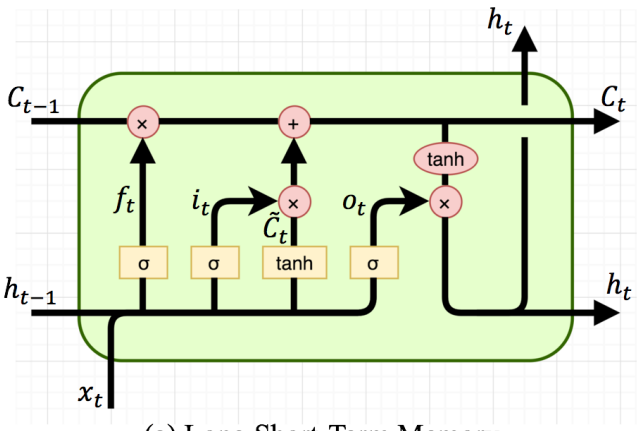
**Dimensions:** The $c^{<t-1>}$, $c^{<t>}$, $a^{<t>}$, $c\_new\_candidate^{<t>}$, $\Gamma_i^{<t>}$, $\Gamma_f^{<t>}$ and $\Gamma_o^{<t>}$ are vectors with identical dimensions.

**Math of LSTM:**

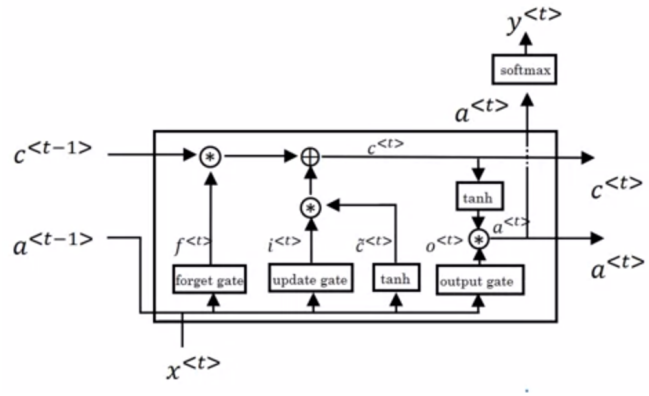
LSTM

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

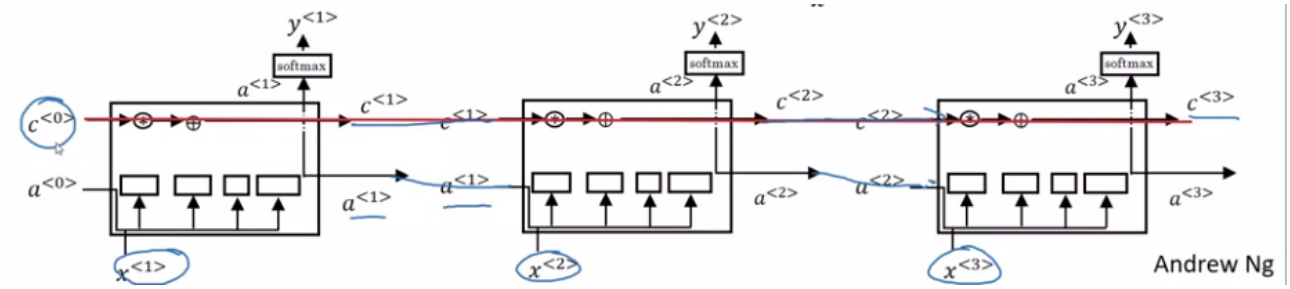
**Cell diagram:**



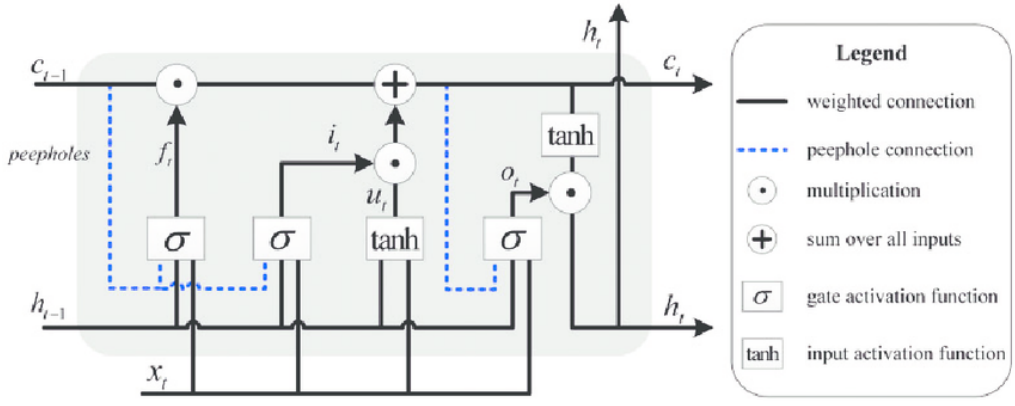

(a) Long Short-Term Memory




It is possible in LSTMs (and also GRUs) to pass on some of the memory cell information that was captured in earlier layers to deeper layers when the forget and update gates are learned appropriately:




**Peephole connections:** LSTM variation that uses peephole connections in which $c^{<t-1>}$ is used also in gates output calculation.



**Bidirectional RNNs (BDRNN)**

The LSTM, GRU and RNN are forward directional only (unidirectional). This means that at any given point in time the output depends only on current time input as well as information from previous layers.

However, there are applications where information from later time steps can also help to make a decision in current time step (e.g. machine translation). For example, in named-entity recognition if we have the following sentences: **a) [He said, "Teddy bears are on sale!"], b) [He said, "Teddy Roosevelt was a great President!"]** and have read until the third word in the sequence it is impossible to know if "Teddy" is a name because the only information we have is from the first 3 words. In order to know if the word "Teddy" is a name or not we need to continue reading the next words. However, if somehow at any given point in time we could have also information from later layers we could know if it is a name or not. This is something that is achieved by BDRNNs: **At a given time we can get information from earlier and LATER parts in the sequence.**

**How BDRNNs operate:** Except the forward $a_f^{<t>}$ activations there are also backward $a_b^{<t>}$ activations. We first compute all the forward activations from left to right and we continue by computing the backward activations from right to left. When all forward and backward activations have been computed the $y^{<t>}$ outputs can also be computed.

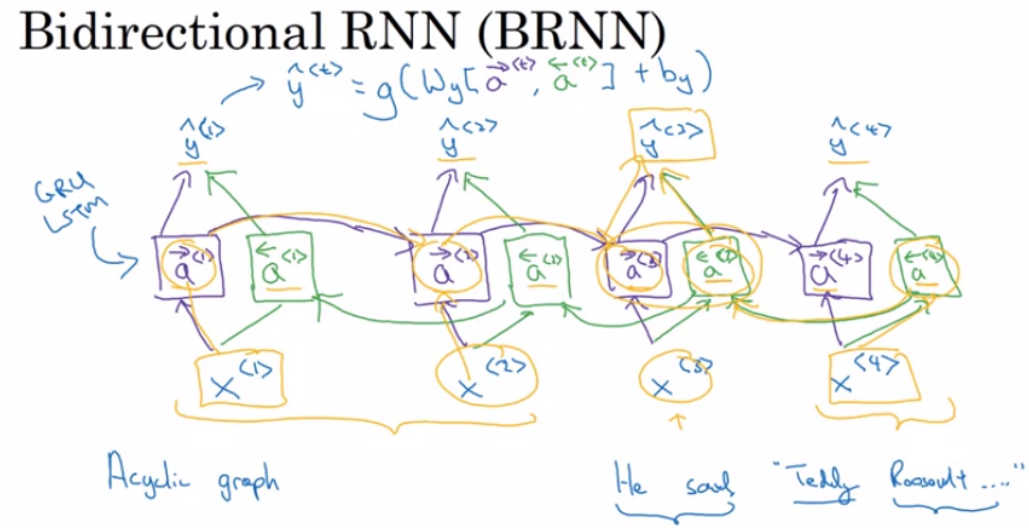
**Disadvantage of BDRNN in real-time applications:** In order to compute the output as well as the loss in BDRNNs we need to read the total sequence and in many real-time applications this might not be effective cause we need to produce some output while the sequence is read.

A BDRNN can use all kinds of cells, e.g. GRU, LSTM, classic RNN cells.

A zero vector is passed as input in the first time step for forward activation and the last time step for the backward activation.

A common scheme in NLP applications is to use a BDRNN with LSTM cells.

**Diagram:**




**Deep RNNs**

To learn very complex functions we can use deep RNNs in which we stack vertically multiple RNN layers. The “deep” refers to vertical extension of the architecture since the sequence models are already deep horizontally.

This architecture can work with the basic RNN as well as the LSTM and GRU cells.

Conventional FF NNs (e.g. MLP, CNN) can be very deep and have many layers. However, in deep RNNs usually we are not using many extra vertical layers because they are already deep due to the temporal dimension.

A variation of a deep RNNs is to connect a sequence of classic FF NN layers on the top of the RNN to generate the $y^{<t>}$ output. However, these layers are not recurrent and are not connected horizontally.

**WOW! We can build a deep version of BDRNNs that uses LSTM cells!**

**Word Embeddings**

**What is it?** A technique to represent feature vectors for words, such that enable NLP learning algorithms (e.g. language models, machine translation, named-entity recognition, etc) to understand analogies/similarities in words and thus operate more effectively as well as generalize better in many cases.

**Generalization difficulty with one-hot word encodings:** Assume that a trained language model exists and predicts successfully the gap word “juice” for the sentence “I want a glass of orange ___”. This doesn’t mean that can also generalize and predict the same word for the sentence “I want a glass of apple/banana/watermelon/(fruit word that does not exist in the training dataset) ___” because any of the words is related to “orange” as any other unrelated word. Actually, the one-hot word vectors as generated by the bag-of-words are orthogonal with 0 inner product (unrelated in the feature space). So, it is difficult for an algorithm to generalize well across different words.

**Wouldn’t be nice for words that are related to some concept to be close in the feature space?** For example, words related to same concept should be close in the feature space, e.g: sexuality (bisexual, homosexual, heterosexual, lesbian, gay), axiom (king, boss, master, chief), colors (black, white, green), numbers (two, four, three), animals (cat, dog, bee), day periods (night, morning, afternoon), gadgets (computer, mp3, camera, smartphone).

**Word Embeddings:** Word embeddings is an algorithm trained on an enormously large corpus of unlabeled text for learning a multi-dimensional feature space as well as the word feature vectors (embeddings) which have the aforementioned properties. So, after learning the word feature vectors, clusters will exist based on some learned concepts. Using these dense word embeddings instead of the one-hot high-dimensional sparse vectors the NLP algorithms can really boost their performance since they can take advantage of words similarities/analogies. For example, in the case of “I want a glass of apple/banana/watermelon ___” more odds will be given to words apple/banana/watermelon because they are closer to already “orange”.

**Embeddings Interpretation:** Although we are learning good word representations, it is difficult to interpret the concepts learned (features) without investigating and visualize the data. It is possible to project in 2D (e.g. using the t-SNE algorithm) the word embeddings in order to investigate which words are similar and why.

**Transfer Learning in NLP applications:**

When we want to build an NLP application and we do not have a large training dataset of textual information – thus our word vocabulary is small – it is very likely that our model will overfit and will not be able to generalize well. The same idea exists in computer vision when we want to train a deep CNN and we have a very small dataset of images. To solve this problem, we usually use generalized pre-trained CNN feature extractors trained on million of images to extract features for our images from small dataset. In general, we transfer knowledge from a task A in a task B only when the knowledge from task A is extracted from a large dataset and task B has a very small dataset. In Keras, there is an Embedding layer which is placed in the beginning of the NN (exactly as the CNN feature extractors) and can be either initialized from a pre-trained word embeddings matrix (to be used in a transfer learning mode) or it can initialize a new one which will be trained together with the rest of the NN. The Embedding layer receives in the input indexes to the vocabulary and generates in the output the word embeddings of the words.

The same idea applies to NLP applications with word embeddings. **We can use pre-trained word embeddings (already extracted feature vectors) which have been resulted from large corpuses of unlabeled text data with 1-100 billions of words.** Thus, we can use the word embeddings for our small dataset's vocabulary. **An NLP application will be able to take advantage of the embedding feature vectors and generalize better.**

For example, here is a training sentence from a named-entity recognition problem: "Sally Johnson is an orange farmer". Even if the model can recognize after training that the first two words are a name it doesn't mean that the same model can also predict correct the name in the following sentences: a) "Robert Lin is an apple farmer" or c) "Robert Lin is a durian cultivator" where the words "durian" and "cultivator" do not exist in the training dataset. However, using pre-trained word embeddings it is very possible that the words "orange/apple/durian" (colors) or "farmer/cultivator" (jobs) will be very close to each other (as feature vectors) and the NLP model **can benefit from it and generalize better**.

**Word analogies emerged from word embeddings:**

One of the emerged properties of word embeddings is **analogy reasoning**. From the learned word embeddings we can find the gap word "Queen" or any other word with similar meaning (assuming that the word exists in the vocabulary) as an answer to the question "Man is to Woman as King is to ___?". To find it we need to solve: $e_{man} - e_{woman} \approx e_{king} - e_{?}$ (the difference between two word embeddings always represent the difference in features between the two words) by finding the word that maximizes the cosine similarity $sim(e_{?}, e_{king} - e_{man} + e_{woman})$. So, word embeddings allow arithmetic operations that have meaning, e.g: $e_{queen} \approx e_{king} - e_{man} + e_{woman}$. **The cosine similarity is very similar to dot product. However, while the dot product takes account both direction and magnitude information from the vectors, the cosine similarity uses only direction information.** Other interesting cases of analogy reasoning: Man:Woman as Boy:Girl, Ottawa:Canada as Nairobi:Kenya, Big:Bigger as Tall:Taller, Yen:Japan as Ruble:Russia.

**Embedding Matrix E:** The word embeddings are discovered by solving an optimization problem using gradient descent. The features are stored in an embedding matrix E of size NxM where N is the number of features to learn and M the number of words in our vocabulary. Each column of the matrix E is the embedding vector of a specific word indexed by the vocabulary word index. The matrix E is initialized randomly and then through optimization w/ gradient descent the features values are learned.

**Algorithms for learning word embeddings**

**1) By training a neural network (e.g. Fully-Connected Feed-Forward) to operate as a language model for predicting the next word from input text context we can learn word embeddings.**

**Extract word vocabulary:** First, using a large corpus of text we extract a candidate vocabulary of words (e.g. 10000 of the most used words by frequency).

**Initialize Embedding Matrix E:** Then, if we want to learn word embeddings with 300 features we initialize randomly an embedding matrix E of size (300, 10000).

**Dataset creation:** We continue by building a dataset of training examples as pairs of (x, y) where y is the next word to predict encoded as a one-hot feature vector and x is the embedding vectors which the algorithm will learn of some context words concatenated as a large input vector. Usually the context words x is an n-gram, the previous n words before the word we want to predict. As an example, if we use a 4-gram text context for x then the input of the NN will be (300x4) since all 4 feature vectors will be concatenated to a single one. The output of the NN will be a softmax of probabilities of vocabulary's size for predicting the next word.

**Training / Optimization:** The NN is trained using backpropagation and gradient descent in order to map the input to output. The key point to learn the word embeddings is to also handle the input features as trainable parameters. So, we can calculate gradients also for the input features and update them with gradient descent. When the NN is trained and has a good generalization then not only we will have trained a language model for predicting the next word but we will also have learned word embeddings for our word vocabulary. The optimization is able to find similar feature embeddings vectors for similar words when various word contexts in the training dataset exist which have small variations and same target output (e.g. if the sentences exist "a glass of orange **juice**" and "a glass of apple **juice**" then the word embeddings of words orange and apple will be similar).

**Variations:** We can also learn word embeddings by training a NN in a similar way using other than n-gram context words. **However, we are not learning word embeddings by training a language model for predicting the next word.** We can use for example as input to the NN both the previous and next n words, or skip-gram: a word around the target word (either before or after it). So, we can experiment with various context words as input and optimize the cost function to find parameters for both: a) the NN to predict the correct target word as well as b) the word embeddings of the vocabulary words. One example of such an algorithm is the Word2Vec algorithm.

**2) Word2Vec**

**a. Skip-gram Word2Vec version**

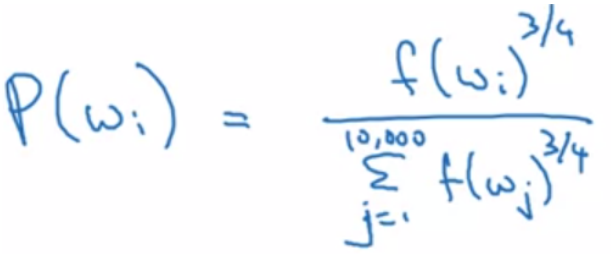
**Extract word vocabulary:** First, using a large corpus of text we extract a candidate vocabulary of words (e.g. 10000 of the most used words by frequency).

**Initialize Embedding Matrix E:** Then, if we want to learn word embeddings with 300 features we initialize randomly an embedding matrix E of size (300, 10000).

**Dataset creation (picking two words with close proximity):** We continue by building a dataset for a supervised learning problem, which consists of **training examples** as pairs of (x, y) where **x is the embedding vector for a random context word and y is another random word as one-hot encoding vector which is close to the x context word (with maximum window range of -/+n**

**words having as middle the context word x)**. So the target word y might be either before or after the word x.

**Sampling the context word x:** When we sample the context word x using the uniform or empirical frequency distribution (probability distribution based on word frequencies) it turns out that the dataset **will have mostly training examples with context words x that are more frequently** (e.g. words such as the, a, an, they, she, he, and, to, etc). This leads to mostly update the embedding vectors of these words and also mostly train the model on these too. In order to balance the common and uncommon words in the sampling we have to use other sampling methods. Here is one example for calculating the probability of each word:




**Model:** A model to map context word x to target word y. Just a shallow softmax classifier trained for multi-class classification: It has one input layer $R^{300}$ and one output layer $R^{10000}$. The operation Wx+b (where parameters W and b are $R^{10000,\ 300}$ and $R^{10000}$ respectively) is calculated in forward propagation to generate the 10000 logits. The logits are used then to generate the softmax output (vector of probabilities for predicting the target word).

**Training / Optimization:** The NN (softmax classifier) is trained using backpropagation and gradient descent in order to map the input to output. **The key point to learn the word embeddings is to also handle the input features as trainable parameters.** So, we can calculate gradients also for the input features and update them with gradient descent. When the NN is trained for predicting the target word we will also have learned some word embeddings for our word vocabulary.

**b. CBOW (continuous bag-of-words) Word2Vec algorithm:**

This architecture is similar to the skip-gram Word2Vec with the only difference that **the target word is the middle word between two context words (one from left and one from right with similar or different distance from the middle word)**. So, there is a difference in the dataset as well as in the size of the input's layer, which is $R^{300x2}$. The embedding feature vectors of the context words are stacked together to a single input feature vector.

**Limitation of methods which use a model softmax output:** When the vocabulary size is enormously large then is very computationally expensive to calculate and sum all the $e^{xi}$ values for the denominator of the softmax's formula:

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

**3) Negative Sampling**

One solution to the problem of softmax heavy computation when having a large vocabulary size is to use **Negative Sampling**. This method hasn't the softmax in the output NN's layer and when the

vocabulary size is large is able to run with a large corpus of text and thus learn very good word embeddings.

**Extract word vocabulary:** Same as Word2Vec.

**Initialize Embedding Matrix E:** Same as Word2Vec.

**Dataset creation:** We continue by building a training dataset of sets where each set contains a positive training example as well as k negative ones relative to the positive one in the set. The k is a hyperparameter which is chosen to be high when we have a small dataset and low when we have a large one. For a specific set j, the positive example is a pair of (x, y) where y is 1 and x is two words: 1) the embedding vector of a random context word from a sentence and 2) a random target word selected from the same sentence which is close to the context word (with maximum window range of -/+n words having as middle the context word x). The k negative examples of the same set j are pairs of (x, y) where y is 0, the context word of x is the same as the one in the positive example and the target word of x is just a random word from the vocabulary.

Here are 2 example sets assuming k=4:

| context word | target word | y |
|---|---|---|
| orange | juice | 1 |
| orange | king | 0 |
| orange | the | 0 |
| orange | water | 0 |
| orange | word | 0 |
| | | |
| blue | sky | 1 |
| blue | she | 0 |
| blue | me | 0 |
| blue | leg | 0 |
| blue | wood | 0 |

When y = 1 it means that the target word is sampled using skip-gram and when y = 0 it means that the target word is sampled randomly from the vocabulary. So, the model has to learn to predict a high P(y=1 | context word, target word) value for positive examples and high P(y=0 | context word, target word) for negative examples in order to distinguish between the two distributions.

**Sampling the context word or the target words in the negative examples:** When we sample the context word using the uniform or empirical frequency distribution (probability distribution based on word frequencies) it turns out that the dataset **will have mostly context words that appear frequently** (e.g. words such as the, a, an, they, she, he, and, to, etc). This leads to mostly update the embedding vectors of these words and also mostly train the model on these too. The target word in positive examples is sampled randomly by skip-gram and there is not problem with it. However, the aforementioned problem can appear even when we randomly sample a target word for the negative examples. In order to balance the common and uncommon words in the sampling we have to different sampling methods. A formula which can be used to calculate the probability of each word can be found in Word2Vec.

**Model:** It has one input layer $R^{300}$ to input the context word and one output layer $R^{10000}$ with 10000 logistic regression units for each possible target word. A theta matrix $\Theta$ exists connecting the input to the output with $R^{10000,\ 300}$ size. Each of the output units estimates P(y=1 | context word x, target

word t) using a logistic regression unit. The activation of an output unit is calculated as σ(np.dot($\Theta_t$, x)+b) where parameter b is a scalar, $\Theta_t$ is a $R^{300}$ parameters vector connecting the input to the output unit which is for a specific target word t and x is the input context word embedding feature vector.

**Training / Optimization:**

Although the output layer of the model has size of $R^{10000}$, when training at each iteration we do not use all the output units. For each iteration we use a subset of the output units depending on the context and target words from the positive and negative examples of the current set. So, the context word x of the set will be used as input and the k+1 target words will specify the output units that will be used. We forward propagate the activations as well as we back propagate gradients in error backpropagation only for the appropriate k+1 output units. It is like training each time a NN with an input layer of size $R^{300}$ and output layer of size $R^{k+1}$. However, on each iteration both the input and the output units are different. The weights connecting the input to the outputs are shared across multiple training iterations.

The NN is trained using backpropagation and gradient descent in order to map the input to output. **The key point to learn the word embeddings is to also handle the input features as trainable parameters.** So, we can calculate gradients also for the input features and update them with gradient descent. When the NN is trained we will also have learned some word embeddings for our word vocabulary.

**4) GloVe (global vectors for word representation)**

It is used rarely but has a good momentum in NLP community.

**Extract word vocabulary:** Same as Word2Vec.

**Initialize Embedding Matrix E:** Same as Word2Vec.

**Dataset creation:** The GloVe algorithm uses **$X_{ij}$ information, which is the number of times the target word i appears in the context of the word j**. So, we need to go through the training corpus of text and calculate it for all of the words in our vocabulary and as an example if the vocabulary contains 10000 words we need to calculate 10000*10000 $X_{ij}$ values. Many of the $X_{ij}$ values will be 0 because some word combinations will not occurred in the corpus text. When skip-gram is used with some relative window of -/+n words having in the middle the context word, **the $X_{ij}$ is a count that captures how often the words i and j appear with each other in the corpus text**. Also, when n is large (e.g. -/+10) then $X_{ij} \approx X_{ji}$. However, this symmetry does not occur when using other contexts, e.g.: when the context word is the previous word of the target word.

Here is an example where $X_{ij} = X_{ji} = 2$:

Assume that the context is a window of -/+4 words having the context word in the middle.

The text of the window will be inside the [] brackets.

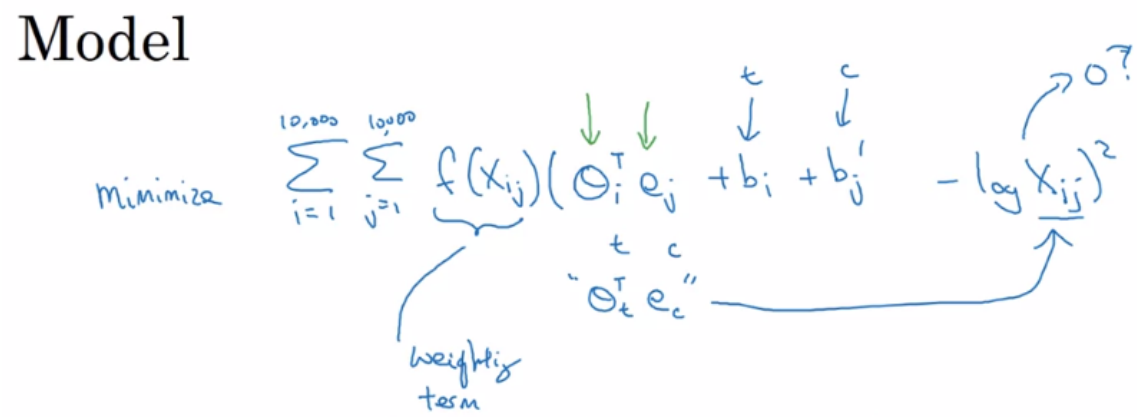
Context word j: **football**
Target word i: *Tuesday*

[I love to play **football** when *Tuesday* is boring] (Tuesday is after one word)
I [love Tuesday to play **football** with my friends] (Tuesday is before two words)

Context word j: **Tuesday**
Target word i: *football*

I love [to play *football* when **Tuesday** is boring] (football is before one word)
[I love **Tuesday** to play *football* with] my friends (football is after two words)

**Model:**




**Training / Optimization:**

We minimize the cost function which is a sum of weighted squared differences from many context-target word pairs (all possible combinations for the words of the vocabulary). When the cost is minimized the various differences are minimized and **the model is trained to approximate how often a specific context-target word pair appears in the corpus text**. The ground truth of how often a context-target word pair appears is the $X_{ij}$. We take the **log of $X_{ij}$ so that all the differences can have the same order of magnitude**. If we don't do it, most of the cost quantity will be because of the difference in context-target word pairs that appear frequently and the optimization will tend to optimize most these. **Also**, we must ignore context-target word pairs which have $X_{ij} = 0$ (pairs that do not exist in the training dataset) because log(0) is -infinity. The way we do it is by multiplying the squared difference with a weight term as a function of $X_{ij}$. So, when $X_{ij} = 0$ then the weight is 0 and 0*(-infinity)=0. **Also**, the weight function is a **heuristic function** that tries to give higher weight to context-target word pairs with small $X_{ij}$ and smaller weight to context-target word pairs with large $X_{ij}$. This can also help the optimization to not tend to optimize mostly the context-target word pairs that appear more frequently but optimize approximately equally for all the context-target word pairs. Both $\Theta_i$ and $e_j$ are symmetric and the final $e_{final}$ for a word can be $(\Theta_i + e_j) / 2$. Lastly, the dimensions of the feature space of the word embeddings are not orthogonal and this make makes very difficult the interpretation of the features, e.g. a feature may not describe one simple property but it can be a combination of many. The model is trained using backpropagation and gradient descent. **The key point to learn the word embeddings is to also handle the input features as trainable parameters.** So, we can calculate gradients also for the input features and update them with gradient descent.

**Sentiment analysis using word embeddings**

A model that solves the task of looking of a piece of text and telling if someone likes or dislikes the thing they are talking about. For example, from a movie or a restaurant review to predict either a rating start number from 1-5 or a text label: liked/disliked/neutral.

For most sentiment classification applications there is a small training labeled dataset and usually the trained model cannot generalize well. The usage of pre-trained word embeddings (generated

from corpus text of billions of words) can help these models to generalize well and operate well with words that are uncommon or do not exist in the training dataset. So, instead of using one-hot encoding vectors or custom word embeddings generated from small corpuses we can use transfer learning of pre-trained word embeddings.

To implement a model and solve the task we can use either a FC-FF or an RNN model:

FC-FF:

A softmax regression classifier is used. The output layer predicts a vector of probabilities for the labels. The input layer is fixed and to handle the fact that the various training examples have variable number of words we take the **sum/average** of all the embedding feature vectors of the words in the sentence and use it as input to the model. The whole model is trained with backprop, gradient descent and cross-entropy cost. The idea is that the sum/average vector will sum/average the meanings of all the words in the training example text.

The approach can work well with training examples such:

“The dessert is excellent” - 4 stars
“The dessert is **outstanding**” - 4 stars (example of generalization with unknown word)
“Service was quite slow” - 2 stars
“Good for a quick meal, but nothing special” - 3 stars

However, there are cases where it is not working well. Look the below example where the sentence has many times the word “good” which means that the sum/average input vector will be affected a lot by it and that the word “lacking” exists only once but its is important that appears first in the sentence:

“Completely lacking in good taste, good service, and good ambience”

The softmax regression classifier will probably classify this sentence as positive, e.g. 4 stars. To solve this problem we need to use a model that takes account the words order and an RNN can do it.

RNN:

We use a many-to-one RNN architecture using as inputs the word embeddings at each time step and at the last time step we output the softmax label prediction. Such a model not only can work well in cases such as “Completely lacking in good taste, good service, and good ambience” but can also generalize well on these because of the word embeddings. For example it can handle also correct the following sentence which uses an unknown word “Completely **failed** in good taste, good service, and good ambience”.

**Debiasing word embeddings**

NLP applications as machine learning products can take important decisions (accept or not a loan applications, accept or not a job application, college admissions) and it is import to use word embeddings that do not have some form of bias. For example, here are some analogy reasoning examples from word embeddings having gender, religion, ethnicity, sexual orientation bias:

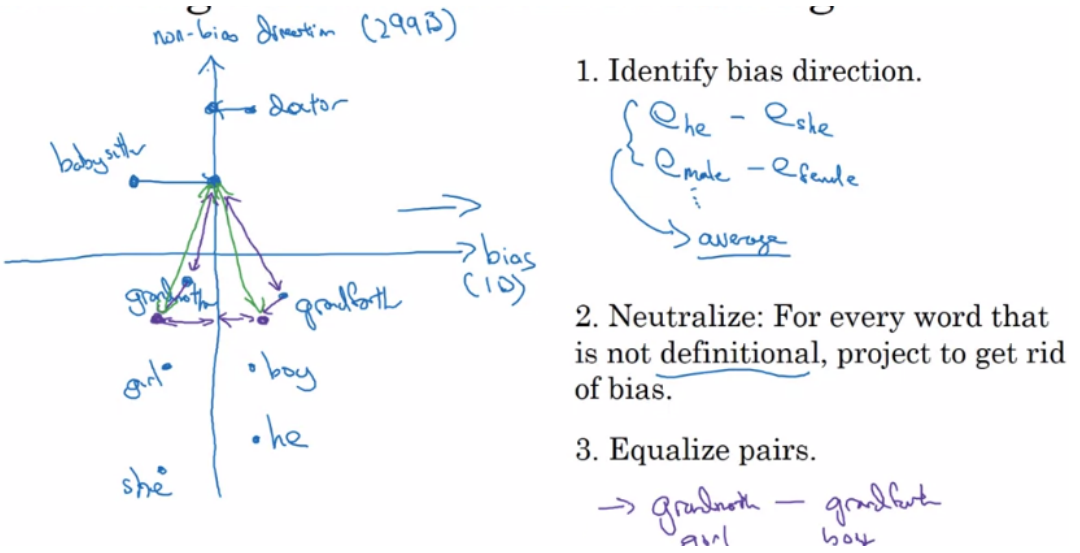
| | | | |
|---|---|---|---|
| Greeks:Superior | as | non-Greeks:Inferior | (ethnicity) |
| Man:Smart | as | Woman:Stupid | (genre) |
| Man:Worker | as | Woman:Homemaker | (genre) |

| | | | |
|---|---|---|---|
| Christian:True | as | non-Christian:False | (religion) |
| Heterosexual:Normal | as | Homosexual/Bisexual:abnormal | (sexual orientation) |

(BTW: many Greeks believe the above today)

So, word embeddings reflect any biases that exist in the text used in the training set. It is important to diminish these biases in the word embeddings so that a Greek or Pakistan, man or woman, Christian or Muslim, straight or gay, can have equal opportunities.

How to debias word embeddings:




The below can be repeated for any other types of bias (religion, ethnicity, sexual orientation, etc).

**1) Identify bias direction:** Assume word embeddings of 300 features already exists. We decide which bias to diminish. Let's assume gender bias. We collect a set of word pairs that have the same meaning but they differ only in gender (e.g. he/she, boy/girl, mum/dad, man/woman, father/mother, grandmother/grandfather, etc). If we had a single words pair from one of the above and we knew that the embedding vectors of the two words differ only in the genre feature we could just have subtracted the two vectors to identify the genre bias direction. However, in practice this never happens when learning word embeddings. That is why we take an average of all the differences of the above word pairs as avg($e_{he} - e_{she}$, $e_{boy} - e_{girl}$, $e_{mum} - e_{dad}$, $e_{man} - e_{woman}$, $e_{father} - e_{mother}$, $e_{grandmother} - e_{grandfather}$, etc) to approximate the genre bias direction (1D subspace).

2) **Neutralize biased words:** Now, we need to transform all the words that they shouldn't have genre bias in order to neutralize them and remove (zero-out) any genre information from them. For example, if the word embedding of the word "doctor" has genre information and is related to "male" word or "babysitter" with "female" we need to transform them so that the two words are genre-independent. One way to do this is to project the embedding vectors of the words to the non-bias subspace as described by the rest 299 dimensions (which are orthogonal/unrelated to the genre bias direction).

**3) Equalize pairs:** Lastly, we need to equalize the distance between each neutralized word and the words of a pair such those used in the 1st step of the algorithm (e.g. boy-girl, man-woman, etc). There is no reason for the neutralized words "doctor" or "babysitter" to be closer to "man" or "woman". If that is the case then it means that the word "man" has features related to "doctor" more than "woman" does. Which means that the words "man" and "woman" do not differ only in genre. The words "man" and "woman" should have the same distance to the neutralized words. So, what

we do is we calculate vectors that move the word embeddings of “man” and “woman” in a new place in the feature space so that both now have the same distance from the non-bias direction.

**How to decide which words need to be neutralized by the second step?** We can train a binary classifier that recognizes if a word is gender-specific or non-gender-specific (e.g. grandmother, computer_programmer).

**Sequence-to-Sequence models**

**Machine Translation:** Many-to-many $T_x$ <> $T_y$ RNN architecture.

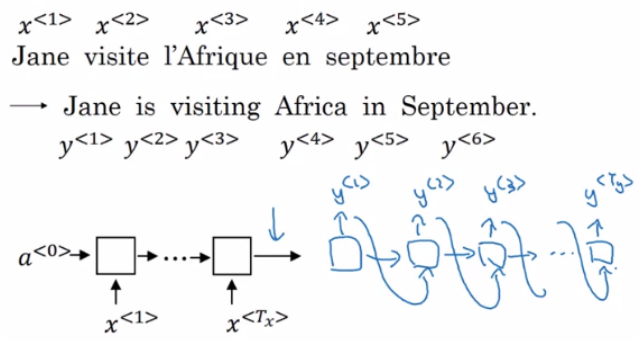
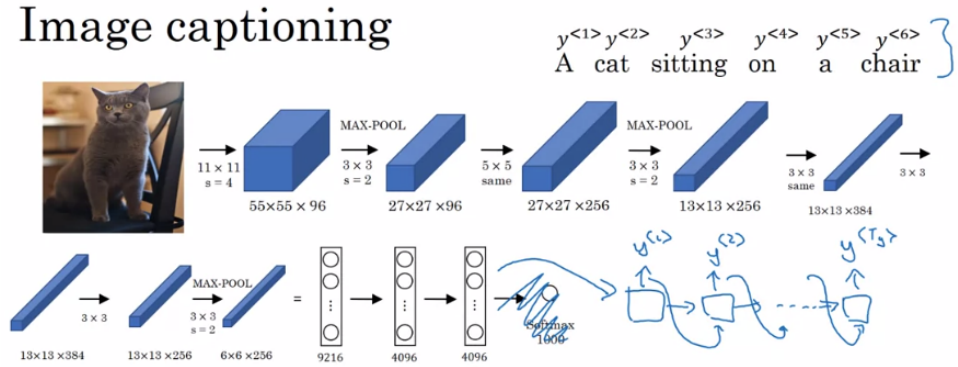



**Image Captioning:** Usually, we combine a CNN with the decoder part of an RNN many-to-many-architecture. The idea is that we want to extract a feature vector from an input image using CNN feature extractors and then feed the feature vector to the decoder to decode it to an output text transcript. So, the initial CNN part of the NN works as an encoder. The CNN can be already pre-trained (state-of-the-art pre-trained CNN architecture for computer vision) or can be trained together with the rest of the NN with backprop and gradient descent. Such an architecture can work well when the output target transcripts are small in length.




**Machine Translation vs Language Model:** They are very similar. The language model tries to predict how much likely is an input sentence $P(y^{<1>}, \ldots, y^{<Ty>})$ with zero input vector (simulating the start of a sentence or a word). However, the many-to-many architecture of machine translation can be seen as a **conditional** language model that instead of getting as input a zero vector it gets the output vector of an encoder and predicts how much likely is an output sentence given the input sequence $P(y^{<1>}, \ldots, y^{<Ty>} \mid x)$. So, we are trying to predict a conditional probability based on a non-zero feature vector (prior knowledge). **Picking the most likely output sequence:** The **language model** can generate text by sampling from the softmax output vector of each time step and passing to the next time step the feature vector (e.g. one-hot word vector or word embedding) that is related with the selected label. We repeat this process a predefined number of times or until we reach the <EOS> token. **However, in sequence models that by design have to generate a sequence based**

**on encoder's output (e.g. machine translation, speech recognition, image captioning) we have to predict the most likely output sequence.** Sampling is not a good solution although it might sometimes generate relevant output sequences. What we want is given an encoded input to predict the sequence output that as a whole maximizes the joint probability. So, it is a search problem. In machine translation the number of all possible translations for a given input text can be exponential large and is not feasible to find the one that is the most likely. Using a greedy search is a better approach than sampling but not the best. Thus we need an optimization algorithm that can approximate the best solution as arg max P(output sequence | input sequence) and **Beam Search** is the answer.

Below is a greedy search bad translation case where the translation is generated step by step and the algorithm chooses the word "going" after the "Jane is" because it has a bigger probability than "visiting". However, the Beam search can manage to find the sequence of words that maximizes the joint probability as a whole.

- "Jane is visiting Africa in September" The correct translation by Beam search
- "Jane is going to visit Africa in September" The one by greedy search

**Beam Search:** We give to the many-to-many RNN an input sequence as input and the encoder encodes it into a vector representation. Then, the vector is given to the decoder part. In the first time step of the decoder we select from the softmax probabilities output vector the B (beam width parameter) most likely output labels. So, instead of taking always the most likely output label as Greedy Search does the Beam Search tries multiples alternatives at the same time so performs a wider search. Using B=1 the Beam Search operates like Greedy Search. We have now estimated $P(y^{<1>} | x)$ and stored the B most likely output labels with their probabilities as 1-token sequence output candidates. We continue to search for more likely bigger sequences. In the next time step of the decoder we estimate $P(y^{<2>} | x, y^{<1>})$ for each of the selected B most likely output labels of the previous time step – except the ones that have as last the <EOS> (end of sequence) token. We compute the joint probabilities $P(y^{<1>}, y^{<2>} | x) = P(y^{<1>} | x) * P(y^{<2>} | x, y^{<1>})$ for the selected B most likely output labels of the previous time step and from all the combinations we select again the B most likely $P(y^{<1>}, y^{<2>} | x)$ with their probabilities as 2-tokens sequence output candidates. We continue that process for some maximum sequence length (e.g. 20 tokens). When we finish the search we have candidate output sequences of different lengths. We select the sequence $P(y^{<1>} | x) * P(y^{<2>} | x, y^{<1>}) * P(y^{<3>} | x, y^{<2>}) * \ldots * P(y^{<N>} | x, y^{<N-1>})$ which is the more likely (higher joint probability).

**Implementation details:** When we implement the Beam Search instead of multiplying bare probabilities we can **sum the log probabilities**. Also, at the end instead of using $P(y^{<1>} | x) * P(y^{<2>} | x, y^{<1>}) * P(y^{<3>} | x, y^{<2>}) * \ldots * P(y^{<N>} | x, y^{<N-1>})$ we can use the **length normalization** formula. All the above implementation details help the algorithm to produce better results and are explained below.

**Length Normalization**

When we try to maximize the joint probability in Beam Search we are actually maximizing a product of probabilities. These probabilities as numbers are < 1 and when we multiply many together the total joint probability can **underflow arithmetically as well as rounding errors can occur with very small probabilities close to zero**. We know that placing a log outside the P(y | x) as log(P(y | x)) it does not affect the maximization and the solution can still be found. This minor addition **can convert the objective from log of product of probabilities to sum of log**

**probabilities** and can help the algorithm because now we log each probability separately, stabilize it and avoid multiplying very small numbers close to zero.

$$\arg\max_y \prod_{t=1}^{T_y} P(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>}) \rightarrow \arg\max_y \sum_{y=1}^{T_y} \log P(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>})$$

Also, when we are trying to estimate a probability P(y | x) and y is a long sequence then it is possible for the probability to be either very small and close to zero (1st formula) or very negative (2nd formula) and this unnaturally tends to promote more likely shorter sequences as output. One way to reduce this unwanted penalty in 2nd formula is to divide the value with $T_y^a$ where $T_y$ is the length of the sequence output and 'a' is a hyperparameter. When a=0 we do not perform any **length normalization** and when a=1 we perform full normalization. Other values for better results can be found with fine-tuning.

$$\frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>})$$

(normalized log likelihood)

**Beam width B:**

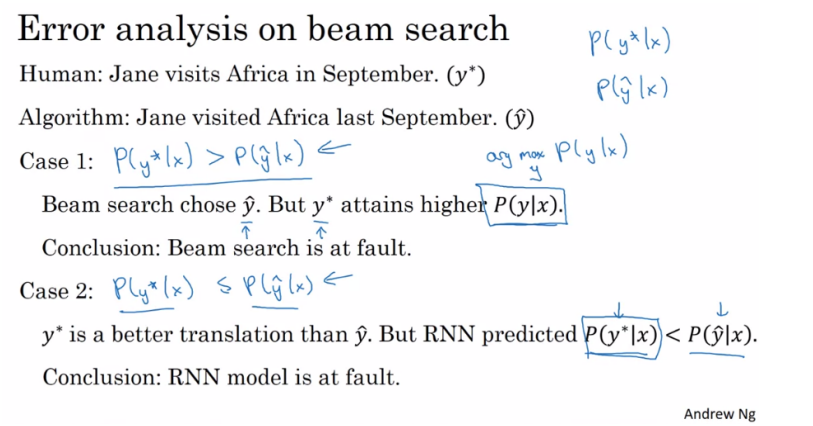
We can start with a small number and try to see any valuable gain from each increasing.

- Large B value: Wide search – evaluate more candidate solutions which leads to better results, More computational expensive, More memory requirements.

- Small B value: Narrow search – evaluate less candidate solutions which leads to worst results, Less computational expensive, Less memory requirements.

**Error analysis with Beam Search**

We can always try to increase the Beam width parameter or train the RNN with more data in order to produce better results. However, it is difficult to know on what to spend time for experimentation. What we can do is the following: Assume a validation set of human translated text examples. If for a validation example (x, y*) the RNN generates P(*y | x) and the beam algorithm generates P(y | x) and P(*y | x) < P(y | x) we can say that the although the RNN gives higher probability to the validation example the Beam search algorithm cannot find it. The blame is on the Beam search algorithm and we might need to increase B width to search wider. If the opposite occurs P(*y | x) <= P(y | x) it means that the RNN couldn't perform well on the human example and Beam search managed to find one that is more or equal likely. However, it would be better for the RNN to estimate the P(*y | x) with a higher probability. So, we can improve the RNN (train on more data, regularization, etc). However, one example is not enough to take decisions on what to improve (RNN or Beam search). We need to perform such an error analysis to the whole validation set and if the most of the validation error is due to bad Beam searching to improve Beam, otherwise improve RNN.

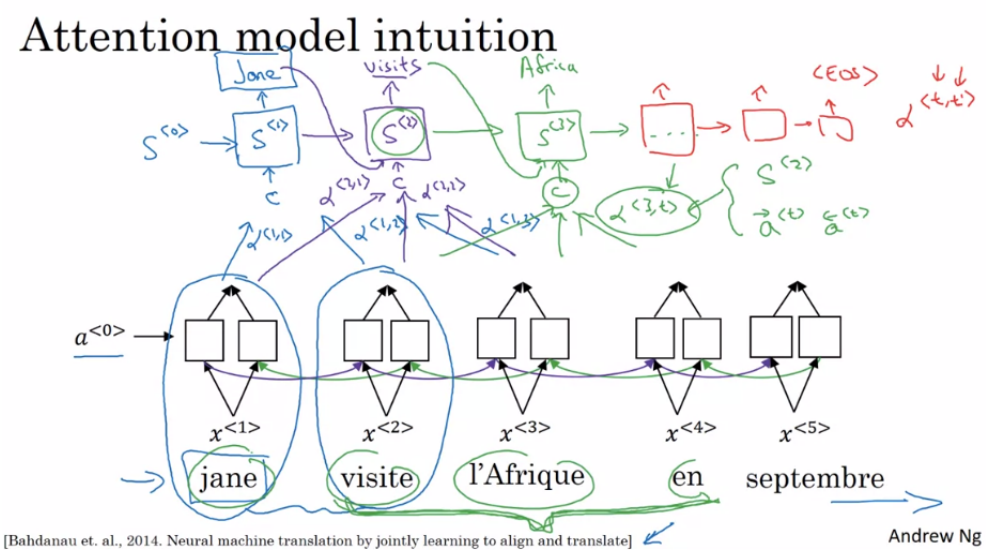


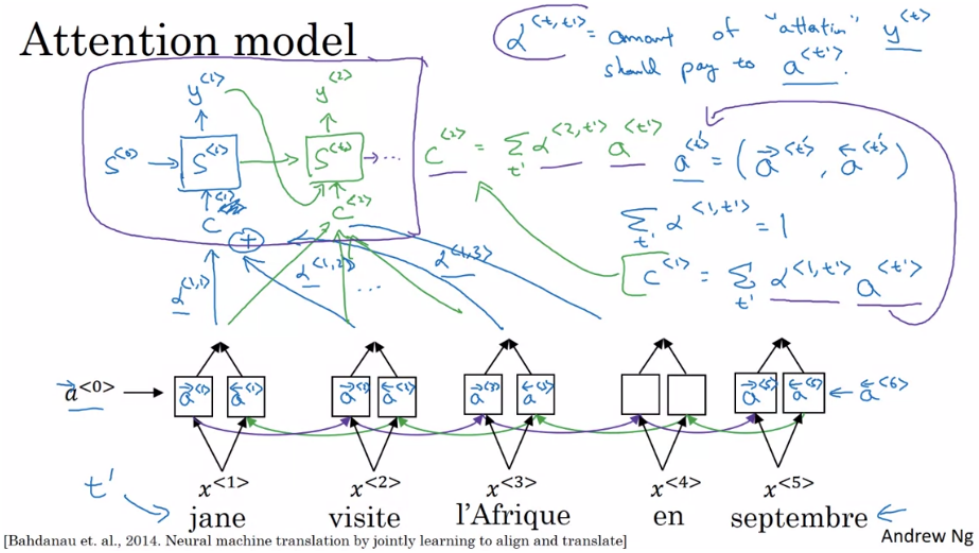

**Attention Model**

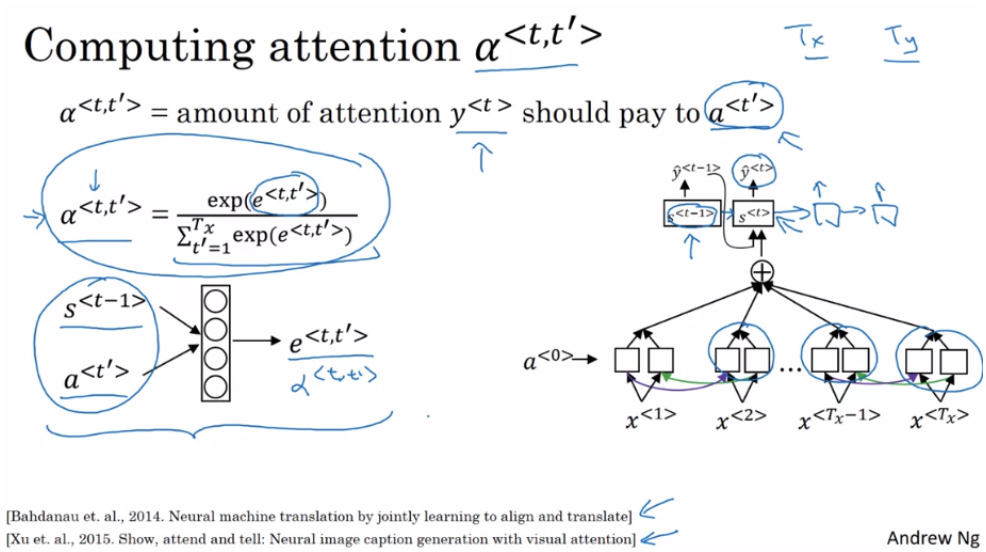
Using a many-to-many RNN architecture (decoder-encoder) with large sequences can be challenging because it cannot memorize information well from long sequences. The performance of an RNN decreases when the length of the input sequences increases. Also, when the input sequences are very small the RNN also performs badly. The GRU and LSTM RNNs can help to solve the vanishing, gradients problem and generate better results with bigger sequences (compared to the basic RNN). However, even better results can be achieved with the Attention Model.

**Machine Translation example with Attention Model:** If we want to use an RNN architecture to learn from very long sentences, the encoder part has to encode the whole input and after the encoding we have to start decoding the internal vector representation with the decoder part. On the other hand, humans do not translate a very long input sentence from a source to a target language after they have read it and memorize it. **But instead they continuously look from left to right with different attention to nearby words and try to generate some translation. This is exactly what Attention Model does!**

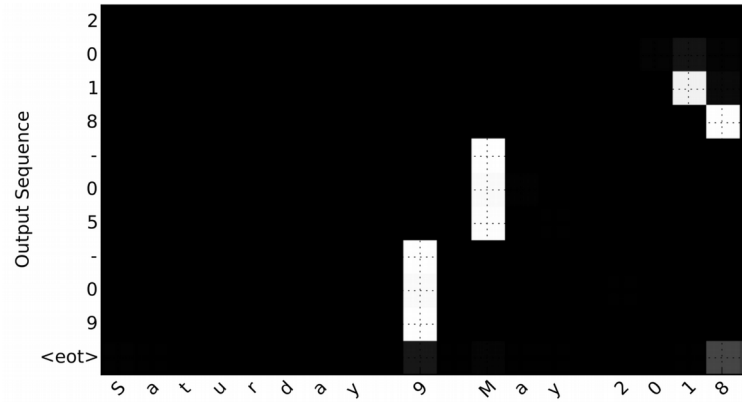
An Attention Model is a combination of a bidirectional RNN (BDRNN) and a unidirectional RNN, both with many-to-many architecture. The RNNs can use the LSTM, GRU or the basic RNN unit. First, we load the input sentence to the BDRNN and generate as output the $a'^{<t>}$ activation ($a'^{<t>}$ is a concatenation of forward and backward activations $a_f^{<t>}$, $a_b^{<t>}$) from each time step which can be seen as a feature vector encoding information from the word of the current time step as well as from previous and after time steps. After all activations are computed for each time step of BDRNN we use the unidirectional RNN to generate word by word the translation until <EOS> token is generated. At each time step of the unidirectional RNN in order to generate the next word we use as input a context which is a weighted sum of BDRNN's $a'^{<t>}$ activations weighted by some "attention" weights $b^{<t, t'>}$. **The intuition is that while training, the unidirectional RNN will learn to pay the appropriate attention to each of the activations from the input context**. In each time step of the unidirectional RNN we generate a new word using a different context. The $b^{<t, t'>}$ weights encode the amount of attention to pay to the input word $a'^{<t'>}$ feature vector in order to generate the $y^{<t>}$ output word. The $b^{<t, t'>}$ weights are not trainable parameters but weight activations operating as coefficients (so we want to be all non-zero and sum to 1, thus softmax is used) generated by trainable layers that tend to learn what to pay attention or not. These layers should be small in size and get as input not only the $a'^{<t'>}$ but also the $s^{<t-1>}$ which is the previous hidden state from the one we already are. See below the images for more information on the formulas.




You can visualize attention weights $b^{<t,\ t'>}$ to see what the network is paying attention to while generating each output:

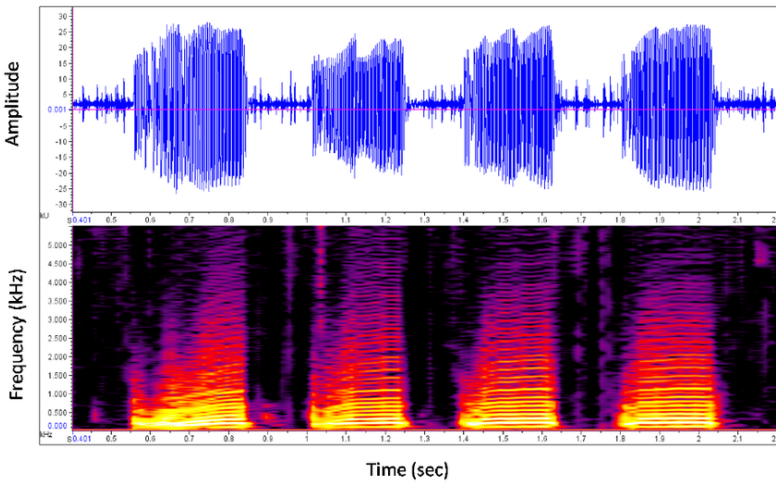


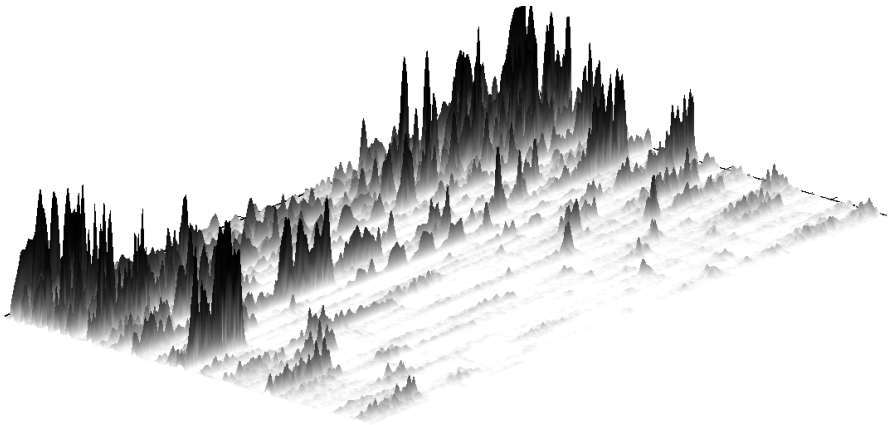

**Application in image captioning:** Can be applied to image captioning where we generate a text description from an input image. However, with the Attention Model at each time we pay "attention" to different parts of the image to generate the next describing word.

**Speech Recognition**

**Task:** A sequence-to-sequence model for mapping an input audio speech clip to a text transcript.

**Audio representation:** Audio by nature is analog, but in digital computers we usually **discretizing** it by sampling with a specific sampling frequency. You can think of an audio recording is a long list of numbers measuring the little air pressure changes detected by the microphone or human ear. For example, if we have a speech clip of 10 seconds and we have sampled it using 100Hz sampling frequency we end up with a **discretized 1D time-series audio signal with 10*100 amplitude values**. So, in speech recognition we operate mostly with **large sequences**. Optionally, the amplitude floating point values can also be digitized by mapping them to a specific range of integer values using a transformation and a bit resolution (e.g. [0-1023] using linear transformation and 10 bits of resolution). An RNN sequence-to-sequence model can be trained with such a representation when it is relatively small. In many applications we have a very large sequence (e.g. with 44.1KHz sampling frequency for 10 seconds we generate 441000 values) and it is very difficult to train the model to learn something. So, another solution is to use **as input the spectrogram of an audio clip which is a 2D time-series of frequencies intensity (how loud is the sound in each frequency)**. At each time step we have a vector of frequency intensities. The spectrogram convolves the raw 1D audio times-series with a window and computes each time the Fourier transform. Depending on the hyperparameters of the spectrogram the 2D time-series output has smaller length than the one of the input 1D time-series.



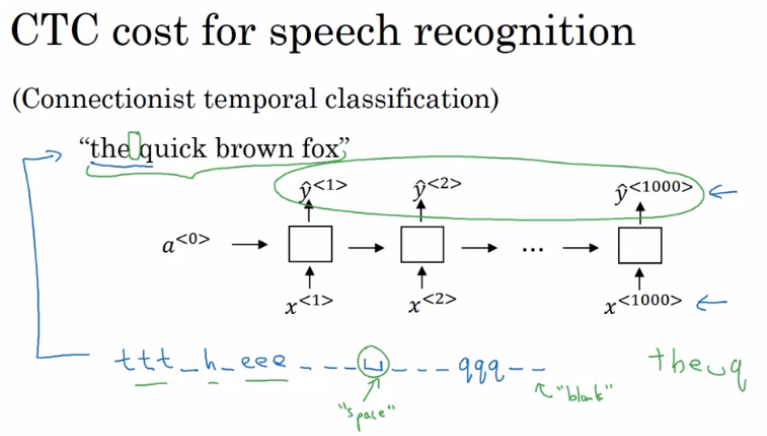


**Phonemes and end-to-end Deep Learning:** Back in time the speech recognition systems were trained with **phonemes** (the basic units of spoken language) which were hand-crafted features. Nowadays, in end-to-end Deep Learning we use large datasets of **transcribed audio** (either as 1D amplitude time-series or 2D spectrogram) and train our models to generate text transcript from audio input.


**Models for speech recognition:**

**Input / Output sequence lengths:** In speech recognition usually the input sequence is much larger (e.g. 10000 values for 10 seconds sampled by 100Hz) than the output sequence (text transcript of alphabet letters).

**Models:** 1) **Attention Model:** Where at each output time step of the unidirectional RNN we are trying to predict the next letter of the text transcription. 2) **Connectionist Temporal Classification (CTC) cost:** A deep BDRNN architecture of LSTM, GRU or basic RNN units can be used. In this architecture the output sequence length is the same as input's. Assume 10000 time steps for 10 seconds of audio sampled with 100Hz sampling frequency. The output of the BDRNN is extremely larger than it needs to be for the output transcription text. So, to handle this the NN is allowed to repeat alphabet character as well as use a blank token "_" which means "nothing", thus can learn to spread out in the output a small text, e.g. "the quick" as "ttt___hhh___eee__<space as single character>__qq__uu__i__cc_kkk" (it can be larger). The generated output sequence is considered as correct by the CTC cost function because it collapses all the repeated characters not separated by the blank token.

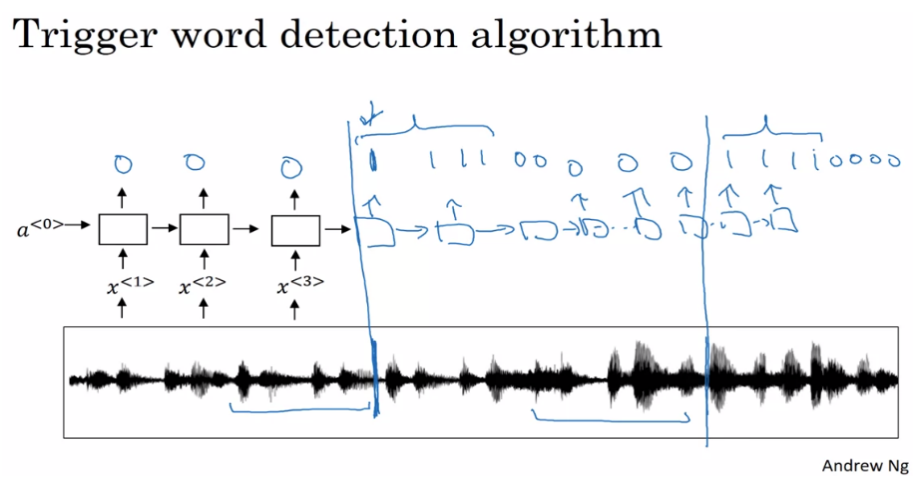



**Trigger Word Detection System**

Wake up a device with a trigger word (Amazon Echo, Baidu DuerOS, Apple Siri, Google Home).

**Model:** A simple many-to-many $T_x = T_y$ RNN can be trained to detect the trigger word.

**Input / Output:** The input of the RNN is an audio signal either as a 1D time-series of amplitude values or as a 2D time-series of frequencies intensity (spectrogram). The output is a sequence of 0s and 1s and we set 1 only in the time step after the trigger word is said. A little hack that can help the model training is to set multiple consecutively 1s (for a short period of time) after the trigger word is said.




Thank you Andrew NG.