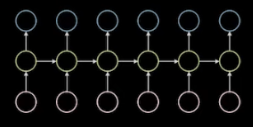
1. **Can you think of a few applications for a sequence-to-sequence RNN? What about a sequence-to-vector RNN? And a vector-to-sequence RNN?**

**Sequence To Sequence RNN**

**First please pay attention that it is different from encode-decoder architecture.**

**One of the example of these models real-time text predictions.**

****

**Seq2Seq**

**Sequence to Sequence Learning with Neural Networks**

**Now we want to check**[***Sequence to Sequence Learning with Neural Networks.***](https://papers.nips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf)

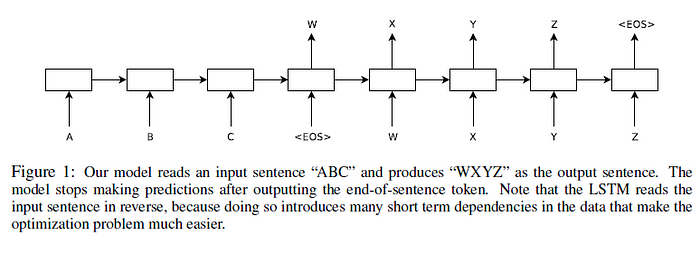
**Authors: Ilya Sutskever, Oriol Vinyals, Quoc V. Le**

**Deep Neural Networks (DNNs) are extremely powerful machine learning models that achieve excellent performance on difficult problems such as speech recognition and visual object recognition because as they can perform arbitrary parallel computation for a modest number of steps.**

**Sequences pose a challenge for DNNs because they require that the dimensionality of the inputs and outputs is known and fixed.**

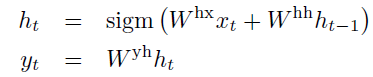
**This paper shows an application of the LSTM architecture which can solve sequence to sequence problems.**

**The idea is to use one LSTM to read the input sequence, one timestep at a time, to obtain large fixed dimensional vector representation, and then to use another LSTM to extract the output sequence from that vector.**

****

**Surprisingly, the LSTM did not suffer on very long sentences, despite the recent experience of other researchers with related architectures. We were able to do well on long sentences because we reversed the order of words in the source sentence but not the target sentences in the training and test set.**

**Here is the general RNN equations:**

****

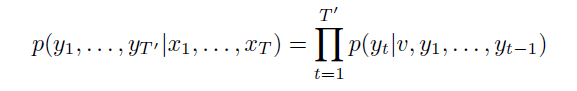
**When the alignment between the inputs and outputs is known ahead of time, the RNN can readily map sequences to sequences. However, it’s unclear how to use an RNN to solve issues with input and output sequences of varying lengths and complex, non-monotonic connections.**

**A basic generic sequence learning method is to use one RNN to map the input sequence to a fixed-sized vector, and then use another RNN to map the vector to the target sequence.**

**It would be difficult to train the RNNs due to the resulting long term dependencies. However LSTM is known to learn problems with long range temporal dependencies.**

**The goal of the LSTM is to estimate the conditional probability p(y1, . . . , yT′ |x1, . . . , xT ).**

**The LSTM computes this conditional probability by first obtaining the fixed dimensional  
representation v of the input sequence given by the last hidden state of LSTM. This is computed using below formula:**

**Our research focused on training a big deep LSTM on a huge number of phrase pairings.  
The training aim is to maximize the log probability of an accurate translation T given the source phrase S.**

**https://miro.medium.com/v2/resize:fit:152/1*OrEcsXzPHlS2anSW7Wk2cg.png**

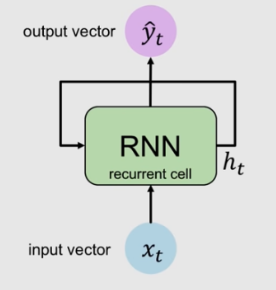
**Once training is complete, we produce translations by finding the most  
likely translation according to the LSTM:**

**https://miro.medium.com/v2/resize:fit:129/1*ql2YzMPwXA7nwfRe299C6g.png**

**We use a basic left-to-right beam search decoder with a modest number B of partial hypotheses, where a partial hypothesis is a prefix of some translation, to find the most likely translation.**

A recurrent neural network (RNN) is a class of [artificial neural networks](https://en.wikipedia.org/wiki/Artificial_neural_network) where connections between nodes form a [directed graph](https://en.wikipedia.org/wiki/Directed_graph) along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Derived from [feedforward neural networks](https://en.wikipedia.org/wiki/Feedforward_neural_networks" \t "_blank), RNNs can use their internal state (memory) to process variable length sequences of inputs.

The memory is the thing that make RNNs sufficient for NLP models such as translation, speech recognition and sentence prediction.



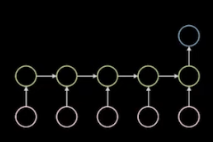
Recurrent neural network

**Vector To Vector RNN**

They just get a simple Vector as their input and produce simple vector as their output. One of their examples can be word to word translation.

**Sequence To Vector RNN**

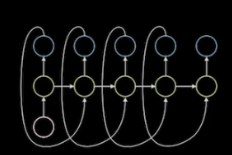
The RNN model, which takes a sequence as input and outputs a single vector, produces a single vector. One of their example is sentence prediction. They look like below:



Seq2Vec

**Vector To Sequence RNN**

The RNN model takes a single vector as input and produces a sequence as output. An example of these models can be image to sentence model, which takes an image(consider it as a vector) and then produces a sentence to describe that image.



Vec2Seq

1. **Why do people use encoder–decoder RNNs rather than plain sequence-to-sequence RNNs for automatic translation?**

Encoder-decoder RNNs excel in translation tasks for their flexibility and efficient handling of varying sequence lengths using a specialized architecture.

Decoder-encoder Because RNNs can accommodate variable input and output sequence lengths, they are frequently chosen for automatic translation over simple sequence-to-sequence RNNs. This gives translators greater freedom when translating across languages with different sentence patterns.

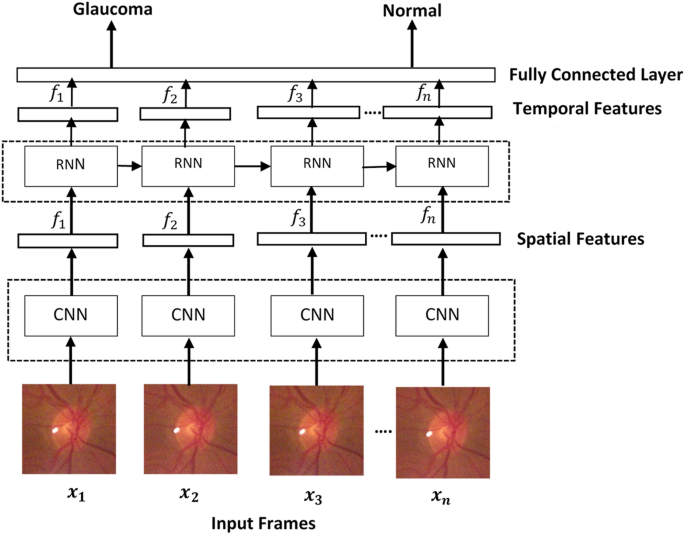
This is achieved through the encoder-decoder architecture, where the encoder processes the input sequence and converts it into a fixed-length context vector, which the decoder then uses to generate the output sequence.

1. **How could you combine a convolutional neural network with an RNN to classify videos?**

### Combining CNN and RNN

We developed a combined CNN (VGG16 and ResNet 50) and RNN (LSTM) architecture to extract spatial and temporal features, respectively. The overall framework of our approach is shown in Fig. [4](https://www.nature.com/articles/s41598-021-81554-4#Fig4). Each video is converted into sequential images and passed onto the CNN to extract spatial features. The outputs are then passed into a recurrent sequence learning model (i.e. LSTM) to identify temporal features within the image sequence. The combined features are finally passed on to a fully connected layer to predict the classification for the full input sequence.

**Figure 4**

[](https://www.nature.com/articles/s41598-021-81554-4/figures/4)

1. **What are the advantages of building an RNN using dynamic\_rnn() rather than static\_rnn()?**

Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) are both powerful deep learning architectures, each with its unique strengths. While CNNs excel in analyzing spatial hierarchies in data, making them the go-to for image processing, RNNs are designed to handle sequential data, offering distinct advantages for sequence classification tasks. Here’s why RNNs often have the upper hand in dealing with sequences:

**Temporal Dynamics Handling:** RNNs are inherently structured to process sequences, whether it's text, time series, or any data where order and context matter. They achieve this through their feedback loops, allowing information from previous steps to persist, which is crucial for understanding the temporal dynamics within a sequence.

**Variable-Length Sequences:** RNNs can handle inputs and outputs of variable lengths thanks to their sequential processing nature. This flexibility is particularly beneficial for tasks like language translation or speech recognition, where the length of the input sequence (e.g., a sentence) doesn't match the length of the output sequence.

**Contextual Information Processing:** The architecture of RNNs enables them to maintain a form of ‘memory’ over the input sequence, allowing them to use context from earlier in the sequence to inform later outputs. This characteristic is vital for understanding the meaning in language-based tasks, where the significance of a word can depend heavily on its context within a sentence or paragraph.

**Efficiency in Parameter Usage:** RNNs share parameters across different parts of a sequence. This sharing means that regardless of the sequence length, the number of parameters remains constant, making RNNs more parameter-efficient than CNNs for long sequences.

**Sequencing and Order Sensitivity:** RNNs are specifically designed to recognize patterns across time, making them inherently sensitive to the order of elements in a sequence. This sensitivity is a significant advantage in tasks where the sequence order of the input data carries critical information, such as time-series forecasting or sentiment analysis in text.

However, it's important to note that RNNs come with their challenges, such as the difficulty in training due to issues like vanishing and exploding gradients. Variants like Long Short-Term Memory (LSTM) networks and Gated Recurrent Unit (GRU) networks have been developed to address these issues, preserving long-range dependencies more effectively.

In presenting RNN projects on your resume or portfolio, highlight the specific advantages leveraged in your work: discuss how you utilized RNNs to process variable-length sequences, capture temporal dependencies, or efficiently manage sequence data. Showcasing your strategic choice of RNNs for sequence classification tasks can underline your deep learning expertise and your ability to match problems with the most appropriate solutions

1. **How can you deal with variable-length input sequences? What about variable-length output sequences?**

# Techniques for handling sequence length in RNNs

**RNNs, or recurrent neural networks, can perform tasks on word, sentence, paragraph, or even document length input.**

**However, this invites a couple of challenges involving the length of the input: (1) how to handle short and variable-length inputs (e.g. sentences of different lengths) and (2) what to do with very long inputs (which are computationally intensive).**

## Techniques for handling input sequences that are too short

**The top rated answer to**[**an SO question on this subject**](https://datascience.stackexchange.com/questions/48796/how-to-feed-lstm-with-different-input-array-sizes)**points to three ways of handling this problem in practice.**

### First approach: mask everything to the same length

**The first and simplest way of handling variable length input is to set a special mask value in the dataset, and pad out the length of each input to the standard length with this mask value set for all additional entries created. Then, create a Masking layer in the model, placed ahead of all downstream layers. Those layers (assuming they support masking) will then proceed to ignore the masked values in their calculations.**

**Note that masking restricts us to layers that support masking. I'm not sure how restrictive this is, however; I assume not very.**

**In the case of an LSTM layer, masking is equivalent to skipping the propogation of the input values through the masked time steps. So a length five vector whose last five layers are masked will only go through the first five layers of an LSTM with 10 time steps. The output of the fifth time step will instead be propogated directly to the output layer.**

### Second approach: set batch size to 1 and have keras mask automatically

**If we set batch\_size to 1 (e.g. we perform stochastic or "live" training) and set the requisite length for the input layer to None, Keras is smart enough to mask your input automatically. This obviously only works if you are OK with batches with just a single sample in them, which is rarely what you want.**

### Third approach: bucketize sequences by length and mask per-bucket

**If the length of the sequence is informative, then perhaps the model will see improved performance with samples batched by size. This feels like a stretch to me, but it does represent another approach to this problem: batch the sentences by length, mask those sentences just-in-time, and feed those batches to the model. This requires building a custom data generator.**

## Techniques for handling long input sequences

**The opposite problem occurs when we have extremely long sequences. Such sequences are a problem because they potentially drive up the time cost of training by heaps and heaps. The "Machine Learning Mastery" blog has**[**a pretty authoritative article**](https://machinelearningmastery.com/handle-long-sequences-long-short-term-memory-recurrent-neural-networks/)**on this. In general, for our purposes, 250 to 500 sequence tokens is the practical limit to maximum sequence length.**

### Naive approaches

**The simplest approach to long sequences is to simply truncate them, usually at the end but potentially at the beginning. This represents data loss, but might not be all that bad.**

**Other approaches are to summarize the sentence by e.g. removing stopwords, or otherwise to randomly pick words out of the sequence.**

### Truncated backpropogation

**"Truncated backpropogation through time" is a complicated name for a simple idea that is relatively in vogue right now: performing gradient updates based on backpropogation passes through a tail-end subset of the LSTM layers. This coarse approach results in less accurate updates, especially to the earlier LSTM time-step layers, but also makes for much faster training times.**

**Truncated backpropogation has the potential to make longer sequences tenable. I will cover this technique in more detail in a future notebook.**

### Encoder-decoders

**Autoencoders can be used to compress the sentence to a smaller space. This would essentially involve training an autoencoder, then feeding the input to just the first half of the model, the constraining half. The output vectors will be mathematically compressed word embeddings according to some loss function, so they will no longer be mappable directly back to the sequence they came from.**

**A step up from autoencoders, both in complexity and accuracy, are sequence-to-sequence RNNs, a new LSTM-based RNN architecture that maps sequences of one length to sequences of another length. Seq-to-seq RNNs have been very successful in translation domains; they power e.g. Google Translate (recall that long Wired article on what happened at Google Translate). The subject of a future notebook!**

1. **What is a common way to distribute training and execution of a deep RNN across multiple GPUs?**

## Why and How to Use Multiple GPUs for Distributed Training

**Data Scientists or Machine Learning enthusiasts training AI models at scale will inevitably reach a cap. When the datasets size increases, the processing time can increase from minutes to hours to days to weeks! Data scientists turn to the inclusion of multiple GPUs along with distributed training for machine learning models to accelerate and develop complete AI models in a fraction of the time.  
We will discuss the usefulness of GPUs versus CPUs for machine learning, why distributed training with multiple GPUs is optimal for larger datasets, and how to get started training machine learning models using the best practices.**

**Why Are GPUs Good For Training Neural Networks?**

**The training phase is the most resource-intensive part of building a neural network or machine learning model. A neural network requires data inputs during the training phase. The model outputs a relevant prediction based on processed data in layers based on changes made between datasets. The first round of input data essentially forms a baseline for the machine learning model to understand; subsequent datasets calculate weights and parameters to train machine prediction accuracy.**

**For datasets that are simple or in a small number, waiting a couple of minutes is feasible. However, as the size or volume of input data increases, training times could reach hours, days, or even longer.**

**CPUs struggle to operate on a large amount of data, such as repetitive calculations on hundreds of thousands of floating-point numbers. Deep neural networks are composed of operations like matrix multiplications and vector additions.**

**One way to increase the speed of this process is to switch distributed training with multiple GPUs. GPUs for distributed training can move the process faster than CPUs based on the number of tensor cores allocated to the training phase.**

**GPUs or graphics processing units were originally designed to handle repetitive calculations in extrapolating and positioning hundreds of thousands of triangles for the graphics of video games. Coupled with a large memory bandwidth and innate ability to execute millions of calculations, GPUs are perfect for the rapid data flow needed for neural network training through hundreds of epochs (or model iterations), ideal for deep learning training.**

**For more details on how GPUs are better for machine and deep learning models, check out our blog post about**[**applications for GPU-based AI and machine learning**](https://exxactcorp.com/blog/Deep-Learning/gpu-based-ai-and-machine-learning-applications)**models.**

## What is Distributed Training In Machine Learning?

**Distributed training takes the workload of the training phase and distributes it across several processors. These mini-processors work in tandem to speed up the training process without degrading the quality of the machine learning model. As the data is divided and analyzed in parallel, each mini-processor trains a copy of the machine learning model on a distinct batch of training data.**

**Results are communicated across processors (either when the batch is completed entirely or as each processor finishes its batch). The next iteration or epoch starts again with a slightly newly trained model until it reaches the desired outcome.**

**There are two most common ways how to distribute training between mini-processors (in our case GPUs): data parallelism and model parallelism.**

#### Data Parallelism

**Data Parallelism is a division of the data and allocating it to each GPU to evaluate using the same AI model. Once a forward pass is complete by all the GPUs, they output a gradient or the model’s learned parameters. Since there are multiple gradients only 1 AI model to train, the gradients are compiled, averaged, and reduced to a single value to finally update the model parameters for the training of the next epoch. This can be done synchronously or asynchronously.**

**Synchronous Data Parallelism is where our groups of GPUs must wait until all other GPUs finish calculating gradients, before averaging, and reducing them to update the model parameters. Once parameters have been updated then can the model proceed with the next epoch.**

**Asynchronous Data Parallelism is where GPUs train independently without having to perform a synchronized gradient calculation. Instead, gradients are communicated back to the parameter server when completed. Each GPU does not wait for the other GPU to finish calculating nor calculate gradient averaging, hence asynchronous. Asynchronous data parallelism requires a separate parameter server for the learning portion of the model so it is a little more costly.**

**Calculating the gradients and averaging the training data after each step is the most compute-intensive. Since they are repetitive calculations, GPUs have been the choice for accelerating this step to reach faster results. Data parallelism is reasonably simple and economically efficient, however, there are times when the model is too large to fit on a single mini-processor.**

#### Model Parallelism

**In contrast to splitting the data, model parallelism splits the model (or workload to train the model) across the worker GPUs. Segmenting the model assigns specific tasks to a single worker or multiple workers to optimize GPU usage. Model parallelism can be thought of as an AI assembly line creating a multi-layer network that can work through large datasets unfeasible for data parallelism. Model parallelism takes an expert to determine how to partition the model but results in better usage and efficiency.**

**Is Multi-GPU Distributed Training Faster?**

**Buying multiple GPUs can be an expensive investment but is much faster than other options. CPUs are also expensive and cannot scale like GPUs. Training your machine learning models across multiple layers and multiple GPUs for distributed training increases productivity and efficiency during the training phase.**

**This means reduced time spent training your models, of course, but it also gives you the ability to produce (and reproduce) results faster and problem-solve before anything gets out of hand. In terms of producing results for your effort, it is the difference between weeks of training versus hours or minutes of training (depending on the number of GPUs in use).**

**The next problem you need to solve is how to start utilizing multiple GPUs for distributed training in your machine learning models**

### How Do I Train With Multiple GPUs?

**If you want to tackle distributed training with multiple GPUs, it will first be important to recognize whether you will need to use data parallelism or model parallelism. This decision will be based on the size and scope of your datasets.**

**Are you able to have each GPU run the entire model with the dataset? Or will it be more time-efficient to run different portions of the model across multiple GPUs with larger datasets? Generally, Data Parallelism is the standard option for distributed learning. Start with synchronous data parallelism before delving deeper into model parallelism or asynchronous data parallelism where a separate dedicated parameter server is needed.**

**We can begin to link your GPUs together in your distributed training process.**

* **Break your data down based on your parallelism decision. For example, you might use the current data batch (the global batch) and divide it across eight sub-batches (local batches). If the global batch has 512 samples and you have eight GPUs, each of the eight local batches will include 64 samples.**
* **Each of the eight GPUs, or mini-processors, runs a local batch independently: forward pass, backward pass, output the weights' gradient, etc.**
* **Weight modifications from local gradients are efficiently blended across all eight mini-processors so everything stays in sync and the model has trained appropriately (when using synchronous data parallelism).**

**It is important to remember that one GPU for distributed training will need to host the collected data and results of the other GPUs during the training phase. You can run into the issue of one GPU running out of memory if you are not paying close attention.**

**Other than this, the benefits far outweigh the cost when considering distributed training with multiple GPUs! In the end, each GPU reduces time spent in the training phase, increases model efficiency, and yields more high-end results when you choose the correct data parallelization for your model.**

### Looking For More Information On Distributed Training and Other Machine Learning Topics?

**Neural networks are highly complex pieces of technology and the training phase alone can be daunting. By utilizing and learning more about how you can leverage additional hardware to create more effective models in less time, data science can change our world! GPUs for distributed training are well worth the initial investment when you can create more effective neural networks in weeks and months instead of months and years.**

**We encourage you to get started on distributed training and deep learning. Check out other articles related to machine learning, distributed training, or best GPUs for neural networks (as well as a plethora of other topics) on our**[**blog.**](https://www.exxactcorp.com/blog)