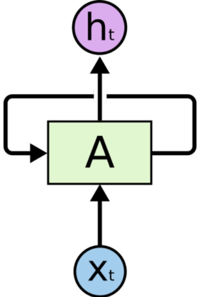
Recurrent Neural Networks (RNN)

Introduction

Humans don't start their thinking from scratch every second. As you read this text, you **understand each word based on your understanding of previous words**. You don't throw everything away and start thinking from scratch again. **Your thoughts have persistence**.

Traditional neural networks can't do this , and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

Recurrent neural networks address this issue, allowing information to persist.



In the above diagram, just one neural network cell, `A`, is depicted. It looks at some input `x_t` and outputs a value `h_t`. In **addition** the that an RNN **allows to loop** information and therefore be passed from one step of the network to the next.

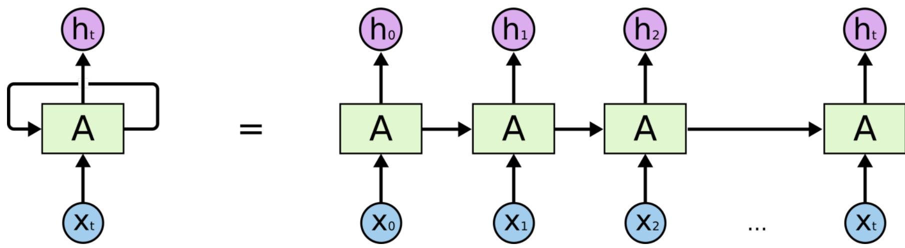
The intuitive general processing equation for an RNN looks the following:

$$h_t = f_W(h_{t-1}, x_t)$$

New State — Some Transferfunction with Parameters W — Old State — Input Data at Timestep t

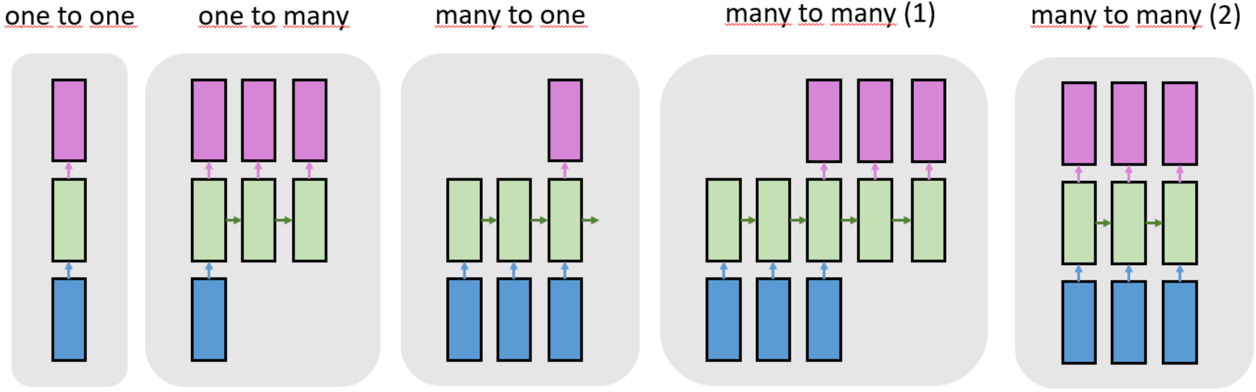
> Note, that it looks the same as in the "*standard*" Feedforward (FFNN) case except that the **output is also a function of some previous cell state/output** . Therefore you can think of RNN as a more general network structure and FFNN just being a special case of RNN. □ (We will see what's up with that in just a moment)

These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we **unroll the loop in time**:



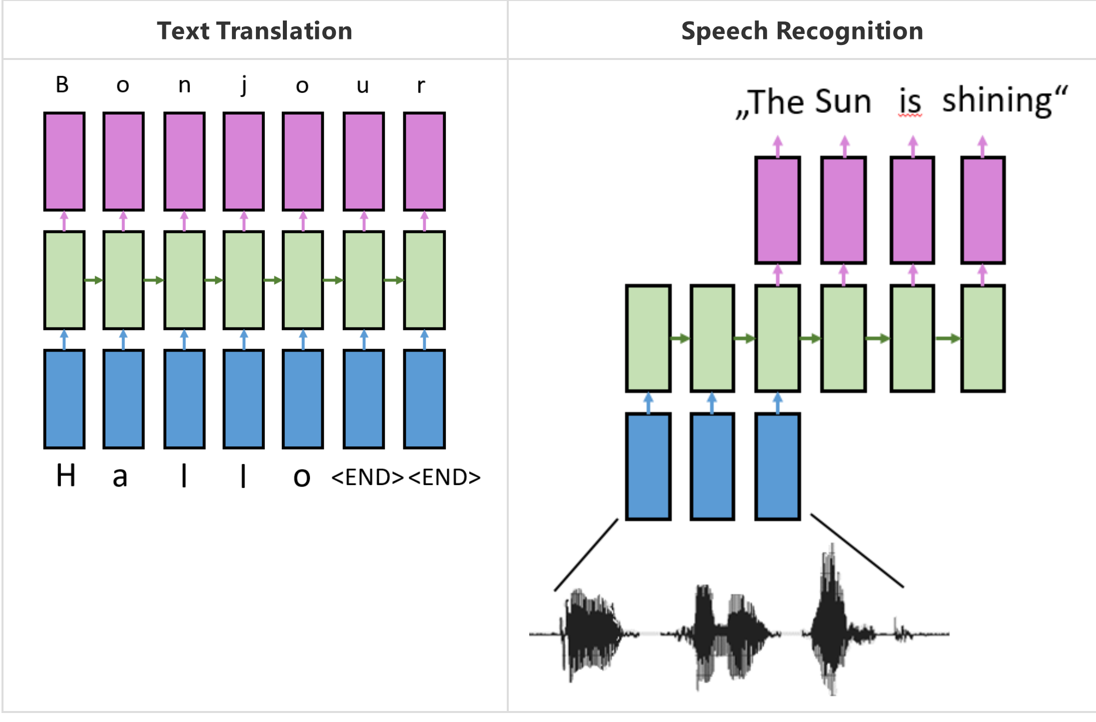
This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.

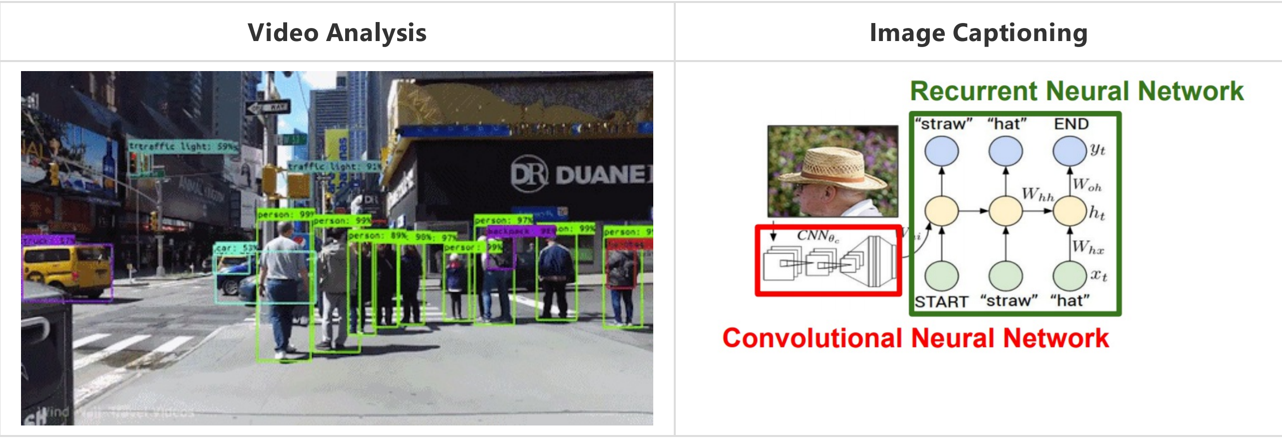
If one looks what we have explained so far, the **unrolling step hints a new freedom domain while designing** neural networks architectures. This leads in general to **five** different possibility's building up RNN Networks:



> There are also special structures like RNN-Autoencoders, where it is sometimes hard to recognize one structure depicted above, but they are usually just a combination of multiple standard models from above. In the Autoencoder case the encoder equals **many to one** and the decoder **one to many**. Therefore the RNN-Autoencoder is **many to one (1)**

In general all of those architectures got their distinguished use cases. And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems:



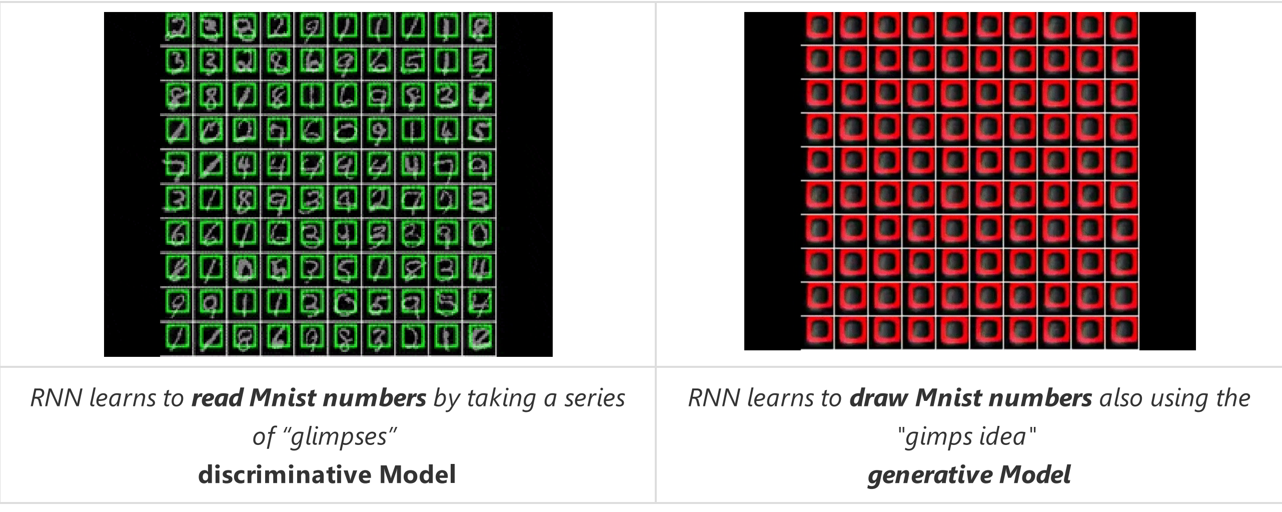


RNN on non Sequence Data

> ***"Can we use RNN's only, to also cover of Non-Sequence Data?"***

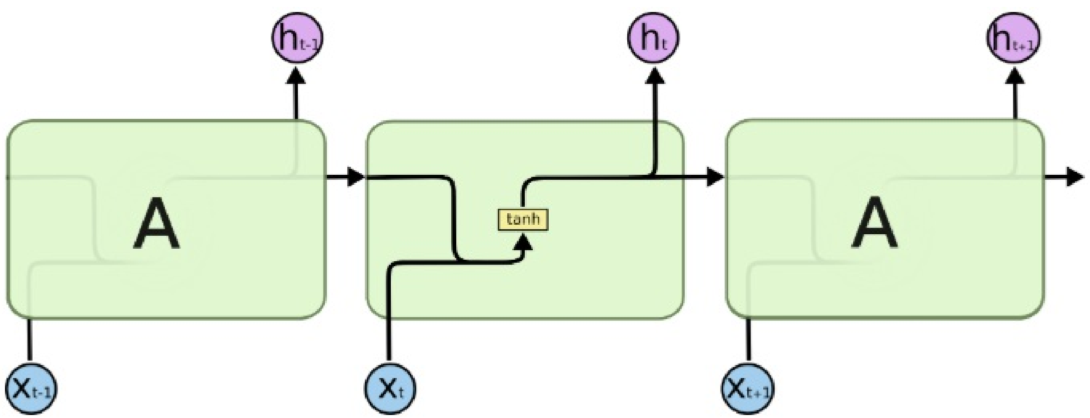
Sure! □

For example, we can recognize the **one to one network** as the **standard FFNN**, we have been previously used to recognize digits in pictures or even draw them.



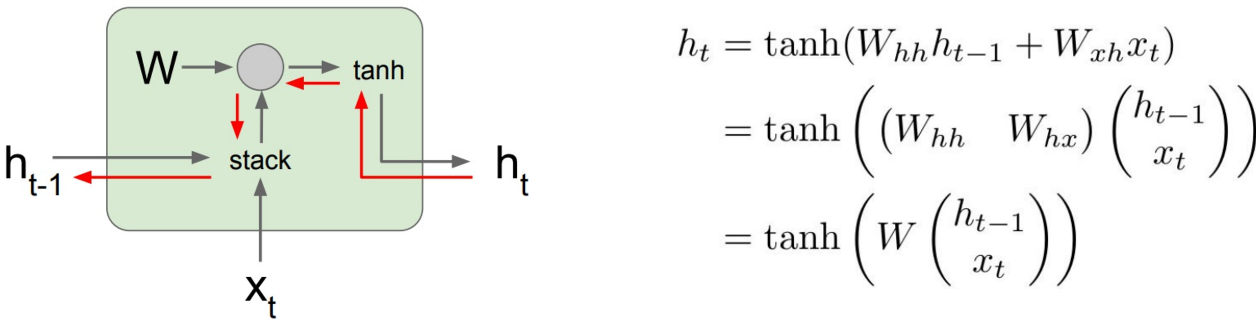
*RNN learns to **read Mnist numbers** by taking a series of "glimpses"* **discriminative Model**	*RNN learns to **draw Mnist numbers** also using the "gimps idea"* ***generative Model***

Vanilla RNN in Detail



Enough Introduction, let's get dirty! 🔲

In the previous section we have seen, that an RNN unrolled in time, like in the picture above, takes input values `x_t` and converts them into his cell state `h_t`. This is calculated as a with respect to a Transfer function (in general tanh), a **learnable** weight matrix and the previous cell state `h_t-1` that it gets to update every time step function is called.



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$
$$= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$

Here is an minimal python implementation of the forward pass step function in a Vanilla RNN:

```
class RNN:
# ...
def step(self, x):
# update the hidden state
    self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
    # compute the output vector
    y = np.dot(self.W_hy, self.h)
    return y
```

You can **recognize the equations** from above in **line 5** and **line 7**. Thankfully we don't have to define the step functions and do the routing between different instances of RNN-Cells ourselfs. 🔲

```
model = Sequential()
model.add(RNN(hidden_size, input_shape=(1, look_back), return_sequences=True)
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
```

Keras does all the Magic for us, defining the Feedforward path in just one line of code (line 2). Here we constructed a shallow RNN of one hidden layer with `hidden_size` cells and `look_back` timesteps of our data.

The Term `return_sequences=True` tells Keras to produce an output in the output layer at every time step, which is important if we want to construct an **one to many** or **many to many** model. The Dense layer is just the layer used

while doing regression tasks (that's why we use `mean_squared_error` as loss for the model, too). Just replace depending on the task you would like to do with our extracted RNN knowledge, e.g. Softmax and Cross-Entropy Loss in a classification task.

Of course we can stack multiple RNN-Cells on top of each other like we did in the deep models we have seen so far. Everything we need to do for adding another RNN-Layer in Keras is:

```
model = Sequential()
model.add(RNN(hidden_size, input_shape=(1, look_back)))
model.add(RNN(hidden_size))
model.add(Dense(1))
model.compile(loss='mean_squared_error',
optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1,
verbose=2)
```

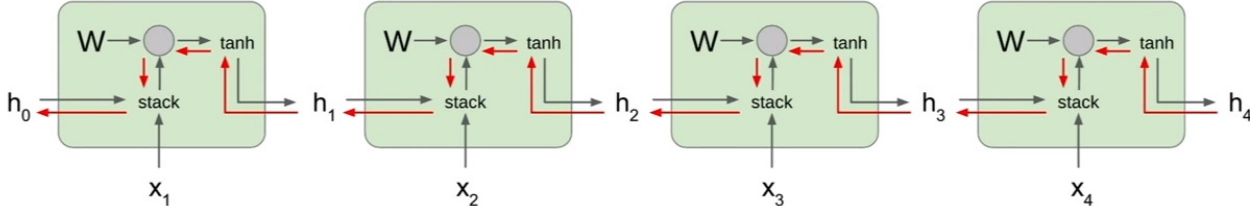
But **stacking high numbers** of RNN-Cells is usually very **uncommon**, due to a problem in the gradient flow of the backward pass of very deep networks.

The so called **Vanishing/Exploding Gradient Problem**. To understand the problem in deep RNN networks we have to understand how the RNN loss is computed.

Vanilla RNN Loss

Backpropagation Through Time (BPTT)

In general the loss of an RNN architecture is exactly the same as in the FFNN case, except of the addition, that it is not only back propagated between layers, but also through time, which is called **Backpropagation Through Time (BPTT)**.

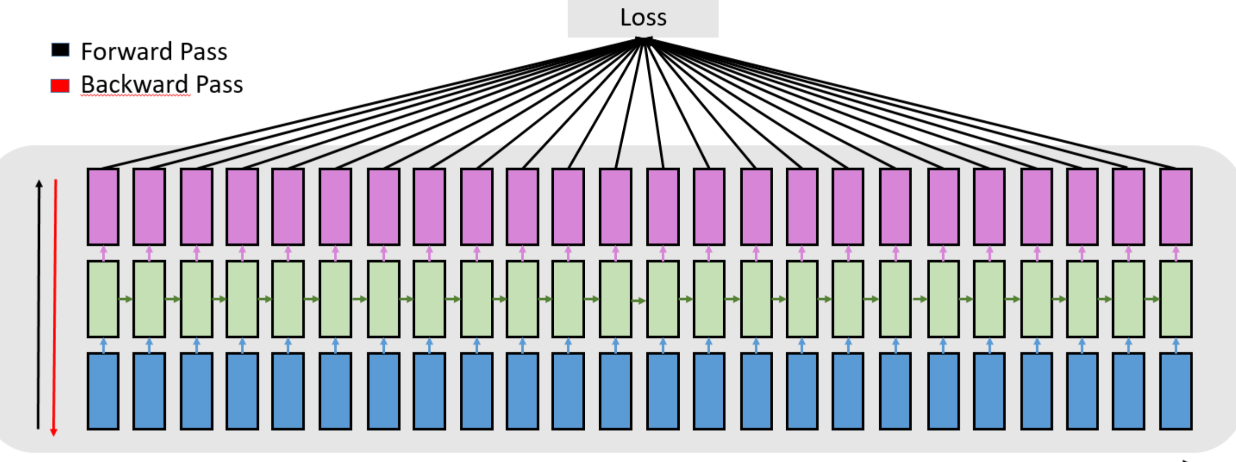


For one cell we run forward through entire sequence to compute loss, then backward through the entire sequence to compute gradient

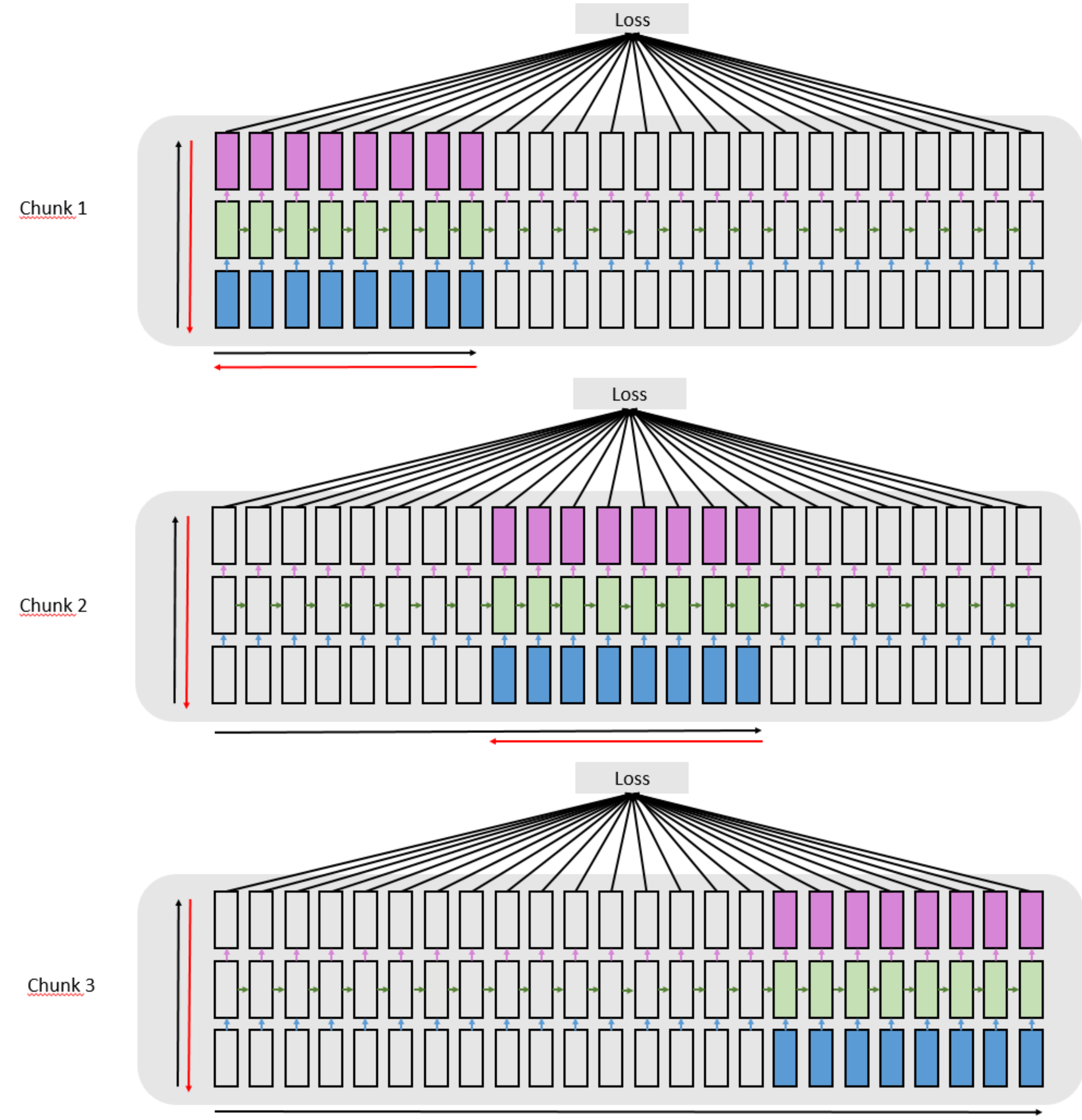
Spatially, each timestep of the **unrolled recurrent neural network** may be **seen as an additional layer** and the internal state from the previous timestep is taken as an input on the subsequent timestep.

Truncated Backpropagation Through Time (TBPTT)

BPTT can be computationally expensive as the number of timesteps increases.



Truncated Backpropagation Through Time (TBPTT), is a modified version of the BPTT training algorithm for recurrent neural networks where the sequence is processed periodically through chunks of the sequence (`k1` timesteps). Therefore the hidden States are carried forward forever but the the BPTT update is performed back for a equal or smaller fixed number of timesteps (`k2` time steps).



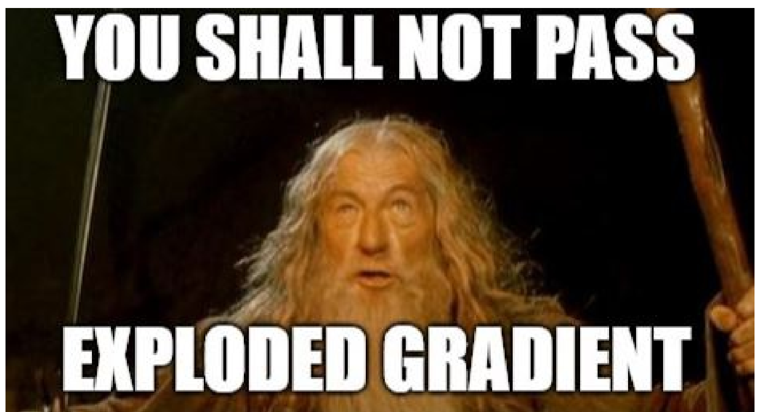
Vanishing/Exploding Gradient Problem

Beeing **computationally expensive** while training is **not the only problem** of **deep RNN Networks**.

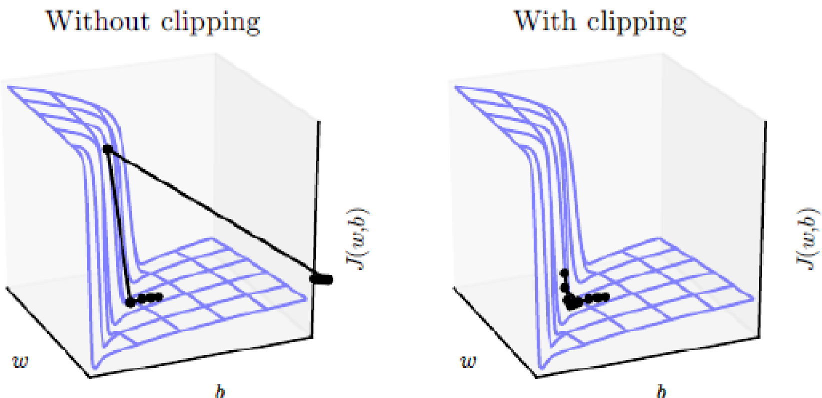
If input sequences are comprised of **thousands of time steps**, then **this will be the number of derivatives required for a single weight update**. This results in two phenomena:

1. **Exploding Gradient**
   If those derivatives are `>1`, this will cause the weights **explode** (go to overflow) and make the model noisy, resulting in eventually overshooting local minima.

   A solution to the overshooting problem would be **Gradient clipping**. Therefore we clip the gradients if its norm is too bigger that some threshold:




```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
   grad*=(threshold / grad_norm)
```



2. **Vanishing Gradient**
   If those derivatives are `<1`, this will cause the weights **vanish** (go to zero) and make learning very slow. Basically the deeper the network the worse it get's and for RNN we **not only have "deepnes" in layers**...

   With that in mind one can understand why RNN are not supposed to go deep in layers, because using RNN is supposed to tackle time related dependencies in the data. **Instead going deep in layers RNN usually go deep in time.**

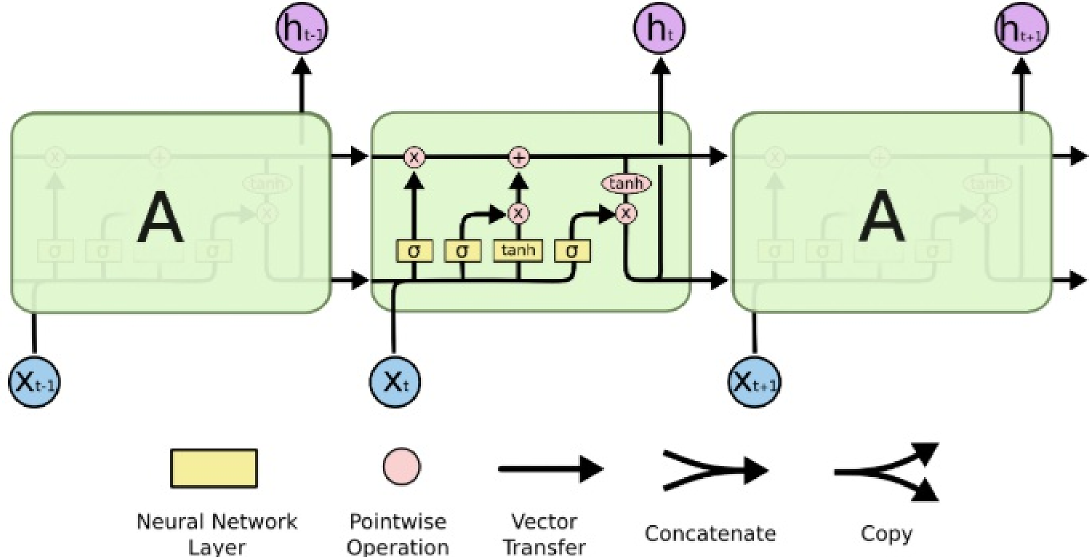
   RNN's are not the only Cell type that suffers from vanishing gradient. In general **all types** of neural Networks using **Sigmoid** or **Tanh** Transfer function **are affected** by the Problem. In FFNN or CNN one can solve the issue, by evoiding Sigm or Tanh, using different functions like **ReLU** instead.

   Unfortunately in RNN using ReLU results in non converging networks, because ReLUs are unbounded above, like **Tanh/Sigmoid** are. The **activations** (values in the neurons in the network, **not the gradients**) can in fact **explode** with extremely deep neural networks like recurrent neural networks. During training, the whole network becomes fragile and unstable in that, if you update weights in the wrong direction even the slightest, the activations can blow up.

   Finally, even though the ReLU derivatives are either `0` or `1`, our **overall derivative** expression contains the **weights multiplied** in. Since the weights are generally initialized to be `< 1`, this could contribute to vanishing gradients. **For this reason, LSTM was invented**.

Long Short Term Memory (LSTM)

Long Short Term Memory , or LSTM, is a structure invented to **tackle the Vanishing Gradient** Problem for RNN. Therefore it introduces a new RNN architecture, the so called ***"Gating Structure"*** depicted below:



The structure is very similar to Vanilla RNN-Networks. But in LSTM two, instead of one, Cell states are preserved and four gates are introduced:

- **f**: Forget gate
- **i**: Input gate
- **g**: Gating gate
- **o**: Output gate
- **c_t**: Cell state, Internal State
- **h_t**: Hidden State, Output State

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Those ***"gating values"*** regulate the information flow between the internal Cell State `c_t` (**Output gate**) and the the Hidden State `h_t`, which is the output given to the next LSTM timestep and the user, similar to the RNN Case.

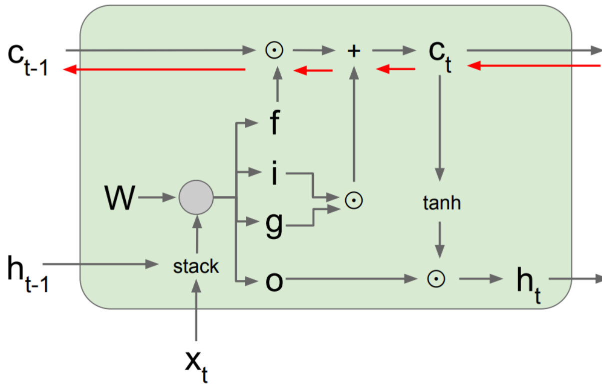
The internal cell information `c_t`, Is calculated using all the other gates, deciding

... how much information from `c_t-1` to keep **(Forget Gate)**
... when ever to write information from `x_t` **(Gating Gate)**
... how much information from `x_t` to capture **(Input Gate)**

> Calling the `h_t` the "hidden" state is sometimes misleading, since the "hidden" state is also propagated as an output, and therefor not tuely hidden. Instead calling the Cell state `c_t` a hidden state would appear more natural, since it is a fully internal (hidden) and therefor hidden in the network. I guess that's what you call a heritage designing LSTM on top of RNN.

The key idea, that handles vanishing gradient in LSTM, is the introduction of some kind of **Gradient Highway**. (Red Arrow) There the gradients can be backpropagated freely just beeing dependent on the previous cell state and the

forget gate output.



Both values are expected to vary in size, not constantly beeing below or above 1 numerically. This makes RNN robust with respect to the Vanishing/Exploding Gradient Problem.

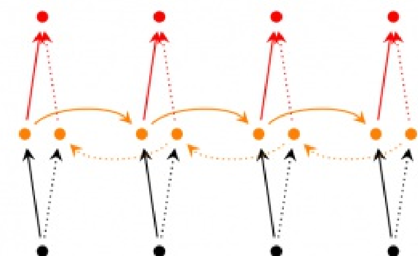
> ### ℹ️ For a detailed proof see this exelent read:
>
> **Bayer, Justin Simon. Learning Sequence Representations. Diss. München, Technische Universität München, Diss., 2015, 2015. - Page 14**

> **Also there is a good quick and dirty explanation, see:**
>
> The vanishing gradient is best explained in the one-dimensional case. The multi-dimensional is more complicated but essentially analogous. You can review it in this excellent paper [1].
>
> Assume we have a hidden state h_t at time step t. If we make things simple and remove biases and inputs, we have
>
> $$h_t = \sigma(wh_{t-1}).$$
>
> Then you can show that
>
> $$\frac{\partial h_{t'}}{\partial h_t} = \prod_{k=1}^{t'-t} w\sigma'(wh_{t'-k})$$
> $$= \underbrace{w^{t'-t}}_{!!!} \prod_{k=1}^{t'-t} \sigma'(wh_{t'-k})$$
>
> The factored marked with !!! is the crucial one. **If the weight is not equal to 1, it will either decay to zero exponentially fast in $t' - t$, or grow exponentially fast**.
>
> In LSTMs, you have the cell state s_t. The derivative there is of the form
>
> $$\frac{\partial s_{t'}}{\partial s_t} = \prod_{k=1}^{t'-t} \sigma(v_{t+k}).$$
>
> Here v_t is the input to the forget gate. As you can see, there is no exponentially fast decaying factor involved. Consequently, there is at least one path where the gradient does not vanish. For the complete derivation, see [2].
>
> [1] Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks." ICML (3) 28 (2013): 1310-1318.
>
> [2] Bayer, Justin Simon. Learning Sequence Representations. Diss. München, Technische Universität München, Diss., 2015, 2015.

Sophisticated RNN Structures

Bidirectional RNNs

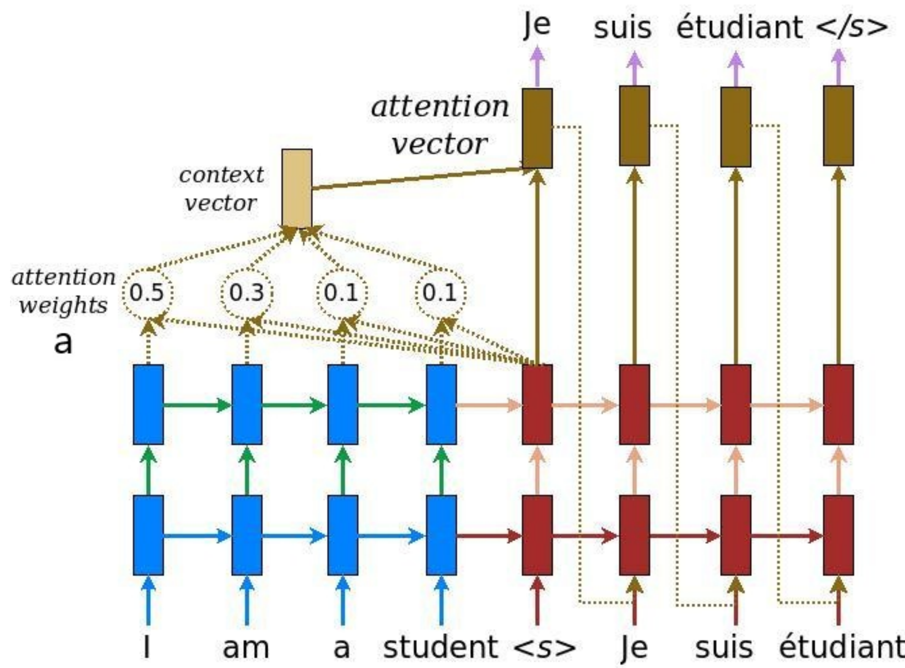
Bidirectional RNNs are based on the idea that the output at time `t` may not only depend on the previous elements in the sequence, but also future elements. For example, to predict a missing word in a sequence you want to look at both the left and the right context. Bidirectional RNNs are quite simple. They are just **two RNNs stacked on top of each other**. The **output** is then computed based on the **hidden state of both RNNs**.



RNN with Attention

Most translation benchmarks are done on languages like French and German, which are quite similar to English (even Chinese word order is quite similar to English). But there are languages (like Japanese) where the last word of a sentence could be highly predictive of the first word in an English translation. **Short: The structure of Language can be very different**. In that case, reversing the input would make things worse. So, what's an alternative? **Attention Mechanisms.**

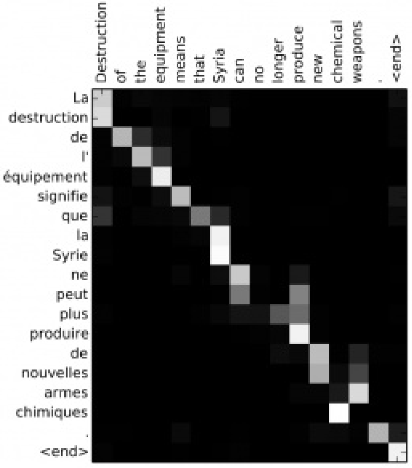
With an attention mechanism we **no longer** try **encode** the full source sentence **into a fixed-length vector**. Rather, we **allow** the decoder **to "attend" to different parts of the source sentence** at each step of the output generation. Importantly, we let the model learn what to attend to based on the input sentence and what it has produced so far. So, in languages that are pretty well aligned (like English and German) the decoder would probably choose to attend to things sequentially. Attending to the first word when producing the first English word, and so on. That's what was done in Neural Machine Translation by Jointly Learning to Align and Translate and look as follows:



Here, The y's are our translated words produced by the decoder, and the x's are our source sentence words. The above illustration uses a 2-ayer RNN, but that's not important and you can just ignore the size of the model, just remind that the input layer is not shown in this picture, but the output layer is (dark brown squares).

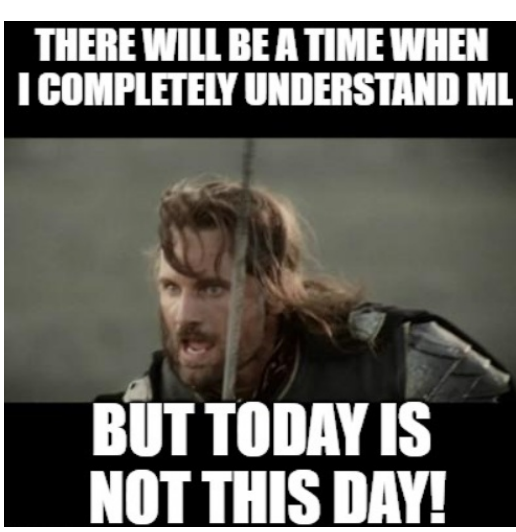
The important part is that each decoder output word `y_t` now depends on a **weighted combination of all the input states** and the last state, <u>not just the last state</u>. The attention weights `a` are weights that define in **how much** of each input state **should be considered for each output**. So, if `a_{1,2}` is a large number (about 0.5 in the picture) , this would mean that the decoder pays a lot of attention to the second state in the source sentence while producing the third word of the target sentence.

A big **advantage of attention** is that it gives us the **ability to interpret** and **visualize** what the model is doing. For example, by visualizing the attention weight matrix a when a sentence is translated, we can understand how the model is translating:



Here we see that while translating from French to English, the network attends sequentially to each input state, but sometimes it attends to two words at time while producing an output, as in translation "la Syrie" to "Syria" for example.

Now your probably like:



<u>But we are here to help!</u>

Questions?