# Recurrent Neural Networks

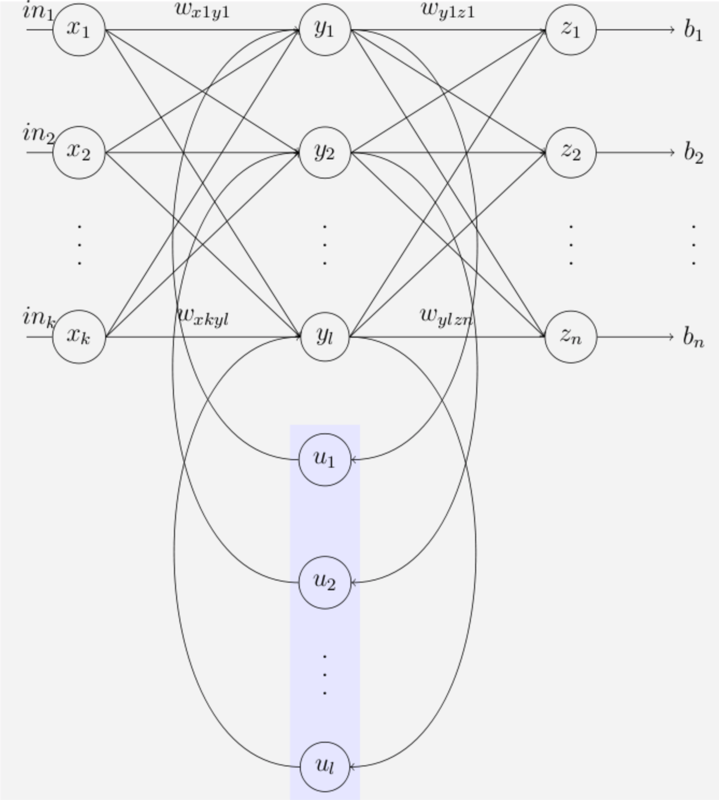
RNNs are based on the same principles as those behind FFNNs, which is why we spent so much time reminding ourselves of the feedforward and backpropagation steps that are used in the training phase.

There are two main differences between FFNNs and RNNs. The Recurrent Neural Network uses:

* **sequences** as inputs in the training phase, and
* **memory** elements

Memory is defined as the output of hidden layer neurons, which will serve as additional input to the network during next training step.

The basic three layer neural network with feedback that serve as memory inputs is called the **Elman Network** and is depicted in the following picture:



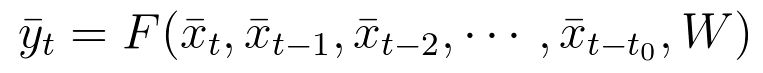
As we've see, in FFNN the output at any time *t*, is a function of the current input and the weights. This can be easily expressed using the following equation:

A picture containing table, clock

Description automatically generated

*Equation 28*

In RNNs, our output at time t, depends not only on the current input and the weight, but also on previous inputs. In this case the output at time t will be defined as:



*Equation 29*

This is the RNN **folded model**:

A close up of a sign

Description automatically generated

*The RNN folded model*

In this picture, *x*¯ represents the input vector,  *y*¯​ represents the output vector and *s*¯ denotes the state vector.

*Wx*​ is the weight matrix connecting the inputs to the state layer.

*Wy*​ is the weight matrix connecting the state layer to the output layer.

*Ws*​ represents the weight matrix connecting the state from the previous timestep to the state in the current timestep.

The model can also be "unfolded in time". The **unfolded model** is usually what we use when working with RNNs.

A picture containing clock, computer, sky, meter

Description automatically generated

*The RNN unfolded model*

In both the folded and unfolded models shown above the following notation is used:

*x*¯ represents the input vector, *y*¯​ represents the output vector and *s*¯ represents the state vector.

*Wx*​ is the weight matrix connecting the inputs to the state layer.

*Wy*​ is the weight matrix connecting the state layer to the output layer.

*Ws*​ represents the weight matrix connecting the state from the previous timestep to the state in the current timestep.

In FFNNs the hidden layer depended only on the current inputs and weights, as well as on an activation function Φ in the following way:

*h*¯=Φ(*x*¯*W*).

In RNNs the state layer depended on the current inputs, their corresponding weights, the activation function and **also** on the previous state:

A picture containing object, clock

Description automatically generated

*Equation 31*

The output vector is calculated exactly the same as in FFNNs. It can be a linear combination of the inputs to each output node with the corresponding weight matrix *Wy*​, or a softmax function of the same linear combination.

*y*¯​*t*​=*s*¯*t*​*Wy*​

or

*y*¯​*t*​=*σ*(*s*¯*t*​*Wy*​)

**Training RNNs**

When we train RNNs we also use backpropagation, but with a conceptual change. The process is similar to that in the FFNN, with the exception that we need to consider previous time steps, as the system has memory. This process is called **Backpropagation Through Time (BPTT)**

For the error calculations we will use the Loss Function, where

*Et*​ represents the output error at time t

*dt*​ represents the desired output at time t

*yt*​ represents the calculated output at time t

A picture containing clock

Description automatically generated

*Equation 35*

In **BPTT** we train the network at timestep t as well as take into account all of the previous timesteps.

The easiest way to explain the idea is to simply jump into an example.

In this example we will focus on the **BPTT** process for time step t=3. You will see that in order to adjust all three weight matrices, *Wx*​,*Ws*​ and *Wy*​, we need to consider timestep 3 as well as timestep 2 and timestep 1.

As we are focusing on timestep t=3, the Loss function will be:

*E*3​=(*d*¯3​−*y*¯​3​)2

A screen shot of a gauge

Description automatically generated

*The Folded Model at Timestep 3*

To update each weight matrix, we need to find the partial derivatives of the Loss Function at time 3, as a function of all of the weight matrices. We will modify each matrix using gradient descent while considering the previous timesteps.

A picture containing meter

Description automatically generated

*Gradient Considerations in the Folded Model*

The unfolded model can be very helpful in visualizing the BPTT process.

A screen shot of a clock

Description automatically generated

The Unfolded Model at timestep 3

**Gradient calculations needed to adjust***Wy*​

The partial derivative of the Loss Function with respect to *Wy*​ is found by a simple one step chain rule: (Note that in this case we do not need to use BPTT. Visualization of the calculations path can be found in the video).

A close up of a logo

Description automatically generated

Equation 36

Generally speaking, we can consider multiple timesteps back, and not only 3 as in this example. For an arbitrary timestep N, the gradient calculation needed for adjusting W\_y*Wy*​, is:

A close up of a logo

Description automatically generated

Equation 37

**Gradient calculations needed to adjust***Ws*​

We still need to adjust *Ws*​ the weight matrix connecting one state to the next and *Wx*​ the weight matrix connecting the input to the state. We will arbitrarily start with *Ws*​.

To understand the **BPTT** process, we can simplify the unfolded model. We will focus on the contributions of *Ws*​ to the output, the following way:

A screen shot of a gauge

Description automatically generated

Simplified Unfolded model for Adjusting Ws

When calculating the partial derivative of the Loss Function with respect to *Ws*​, we need to consider all of the states contributing to the output. In the case of this example it will be states *s*3​¯​ which depends on its predecessor *s*2​¯​ which depends on its predecessor *s*1​¯​, the first state.

In **BPTT** we will take into account every gradient stemming from each state, **accumulating** all of these contributions.

* At timestep t=3, the contribution to the gradient stemming from *s*3​¯​ is the following : (Notice the use of the chain rule here. If you need, go back to the video to visualize the calculation path).

A screenshot of a cell phone

Description automatically generated

Equation 38

* At timestep t=3, the contribution to the gradient stemming from *s*2​¯​ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of *s*2​¯​ to *s*3​¯​ . If you need, go back to the video to visualize the calculation path).

A close up of a sign

Description automatically generated

Equation 39

* At timestep t=3, the contribution to the gradient stemming from \bar{s\_1}*s*1​¯​ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of \bar{s\_1}*s*1​¯​ to \bar{s\_2}*s*2​¯​ and \bar{s\_3}*s*3​¯​ . If you need, go back to the video to visualize the calculation path).

A close up of a sign

Description automatically generated

Equation 40

After considering the contributions from all three states: \bar{s\_3}*s*3​¯​ ,\bar{s\_2}*s*2​¯​ and \bar{s\_1}*s*1​¯​, we will **accumulate** them to find the final gradient calculation.

The following equation is the gradient contributing to the adjustment of W\_s*Ws*​ using **Backpropagation Through Time**:

A screenshot of a cell phone

Description automatically generated

Equation 41

In this example we had 3 time steps to consider, therefore we accumulated three partial derivative calculations. Generally speaking, we can consider multiple timesteps back. If you look closely at the three components of equation 41, you will notice a pattern. You will find that as we propagate a step back, we have an additional partial derivatives to consider in the chain rule. Mathematically this can be easily written in the following general equation for adjusting W\_s*Ws*​ using **BPTT**:

A picture containing object, clock

Description automatically generated

**Gradient calculations needed to adjust***Wx*​

To further understand the **BPTT** process, we will simplify the unfolded model again. This time the focus will be on the contributions of W\_x*Wx*​ to the output, the following way:

A picture containing sky, clock

Description automatically generated

Simplified Unfolded model for Adjusting Wx

When calculating the partial derivative of the Loss Function with respect to to W\_x*Wx*​ we need to consider, again, all of the states contributing to the output. As we saw before, in the case of this example it will be states \bar{s\_3}*s*3​¯​ which depend on its predecessor \bar{s\_2}*s*2​¯​ which depends on its predecessor \bar{s\_1}*s*1​¯​, the first state.

As we mentioned previously, in **BPTT** we will take into account each gradient stemming from each state, **accumulating** all of the contributions.

* At timestep t=3, the contribution to the gradient stemming from \bar{s\_3}*s*3​¯​ is the following : (Notice the use of the chain rule here. If you need, go back to the video to visualize the calculation path).

A picture containing table

Description automatically generated

Equation 43

* At timestep t=3, the contribution to the gradient stemming from \bar{s\_2}*s*2​¯​ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of \bar{s\_2}*s*2​¯​ to \bar{s\_3}*s*3​¯​ . If you need, go back to the video to visualize the calculation path).

A close up of a sign

Description automatically generated

Equation 44

* At timestep t=3, the contribution to the gradient stemming from \bar{s\_1}*s*1​¯​ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of \bar{s\_1}*s*1​¯​ to \bar{s\_2}*s*2​¯​ and \bar{s\_3}*s*3​¯​ . If you need, go back to the video to visualize the calculation path).

A close up of a sign

Description automatically generated

Equation 45

After considering the contributions from all three states: \bar{s\_3}*s*3​¯​ ,\bar{s\_2}*s*2​¯​ and \bar{s\_1}*s*1​¯​, we will **accumulate** them to find the final gradient calculation.

The following equation is the gradient contributing to the adjustment of W\_x*Wx*​ using **Backpropagation Through Time**:

A screenshot of a cell phone

Description automatically generated

Equation 46

As mentioned before, in this example we had 3 time steps to consider, therefore we accumulated three partial derivative calculations. Generally speaking, we can consider multiple timesteps back. If you look closely at equations 1, 2 and 3, you will notice a pattern again. You will find that as we propagate a step back, we have an additional partial derivatives to consider in the chain rule. Mathematically this can be easily written in the following general equation for adjusting W\_x*Wx*​ using **BPTT**:

A picture containing object

Description automatically generated

Equation 47

Notice the similarities between the calculations of \frac{\partial{E\_3} }{\partial W\_s}∂*Ws*​∂*E*3​​ and \frac{\partial{E\_3} }{\partial W\_x}∂*Wx*​∂*E*3​​. Hopefully after understanding the calculation process of \frac{\partial{E\_3} }{\partial W\_s}∂*Ws*​∂*E*3​​, understanding that of \frac{\partial{E\_3} }{\partial W\_x}∂*Wx*​∂*E*3​​ was straight forward.

**Summary**

As you have seen, in RNNs the current state depends on the input as well as the previous states, with the use of an activation function.

A close up of a clock

Description automatically generated

*Equation 56*

The current output is a simple linear combination of the current state elements with the corresponding weight matrix.

*y*¯​*t*​=*s*¯*t*​*Wy*​ (without the use of an activation function)

or

*y*¯​*t*​=*σ*(*s*¯*t*​*Wy*​) (with the use of an activation function)

*Equation 57*

We can represent the recurrent network with the use of a folded model or an unfolded model:

A screen shot of a clock

Description automatically generated

*The RNN Folded Model*

A picture containing text, sitting, monitor, screen

Description automatically generated

*The RNN Unfolded Model*

In the case of a single hidden (state) layer, we will have three weight matrices to consider. Here we use the following notations:

*Wx*​ - represents the weight matrix connecting the inputs to the state layer.

*Wy*​ - represents the weight matrix connecting the state to the output.

*Ws*​ - represents the weight matrix connecting the state from the previous timestep to the state in the following timestep.

The gradient calculations for the purpose of adjusting the weight matrices are the following:

A close up of a logo

Description automatically generated

*Equation 58*

A picture containing object, clock

Description automatically generated

*Equation 59*

A picture containing object

Description automatically generated

*Equation 60*

In equations \_51\_ and \_52\_ we used **Backpropagation Through Time (BPTT)** where we accumulate all of the contributions from previous timesteps.

When training RNNs using BPTT, we can choose to use mini-batches, where we update the weights in batches periodically (as opposed to once every inputs sample). We calculate the gradient for each step but do not update the weights right away. Instead, we update the weights once every fixed number of steps. This helps reduce the complexity of the training process and helps remove noise from the weight updates.

The following is the equation used for **Mini-Batch Training Using Gradient Descent**: (where \delta\_{ij}*δij*​ represents the gradient calculated once every inputs sample and M represents the number of gradients we accumulate in the process).

A close up of a clock

Description automatically generated

*Equation 61*

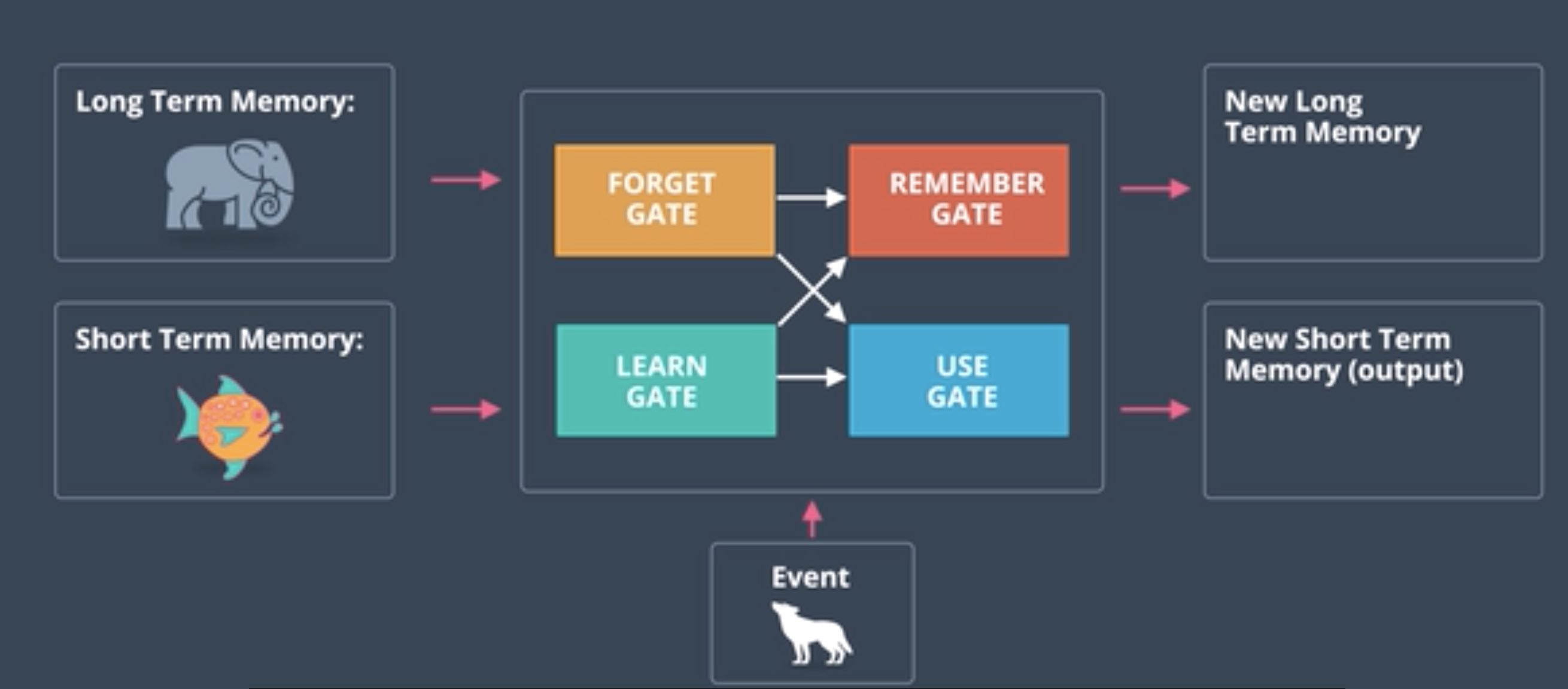
If we backpropagate more than ~10 timesteps, the gradient will become too small. This phenomena is known as the **vanishing gradient problem** where the contribution of information decays geometrically over time. Therefore temporal dependencies that span many time steps will effectively be discarded by the network. **Long Short-Term Memory (LSTM)** cells were designed to specifically solve this problem.

In RNNs we can also have the opposite problem, called the **exploding gradient** problem, in which the value of the gradient grows uncontrollably. A simple solution for the exploding gradient problem is **Gradient Clipping**.

More information about Gradient Clipping can be found [here](https://arxiv.org/abs/1211.5063).

You can concentrate on Algorithm 1 which describes the gradient clipping idea in simplicity.

# LSTM



A picture containing clock

Description automatically generated

A screenshot of a video game

Description automatically generated

**Learn gate**

Combines(tanh) the short term memory and the event and then it ignores(sigmoid) a bit of it keeping the important part. Ignore factor(i-t)

**A close up of a sign

Description automatically generated**

**Forget gate**

Takes the LTM and decides to what to keep and forget. Forget factor(f-t)

A picture containing clock

Description automatically generated

**Remember gate**

Combines the LTM coming from Forget gate and the STM coming from the Learn gate and adds them.

**A picture containing clock, meter

Description automatically generated**

**Use gate**

We will take what is useful from the LTM(U-t) and what is useful from the STM(V-t) to obtain the NSTM

**A screenshot of a cell phone

Description automatically generated**

**Putting it all together**

**A picture containing object, clock

Description automatically generated**

If we see the forget gate the LTM doesn’t have a say in what part of it to keep and what to forget. For that we can have a LSTM with peephole connections by simply combining LTM at all forget nodes(sigmoid)

A picture containing object, clock

Description automatically generated

LSTM sources –(<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>, <http://blog.echen.me/2017/05/30/exploring-lstms/>, <https://www.youtube.com/watch?v=iX5V1WpxxkY> )

GRU – (<http://www.cs.toronto.edu/~guerzhoy/321/lec/W09/rnn_gated.pdf> )

## Define the RNN

## Next, we define an RNN in PyTorch. We'll use nn.RNN to create an RNN layer, then we'll add a last, fully-connected layer to get the output size that we want. An RNN takes in a number of parameters:

* **input\_size** - the size of the input
* **hidden\_dim** - the number of features in the RNN output and in the hidden state
* **n\_layers** - the number of layers that make up the RNN, typically 1-3; greater than 1 means that you'll create a stacked RNN
* **batch\_first** - whether or not the input/output of the RNN will have the batch\_size as the first dimension (batch\_size, seq\_length, hidden\_dim)