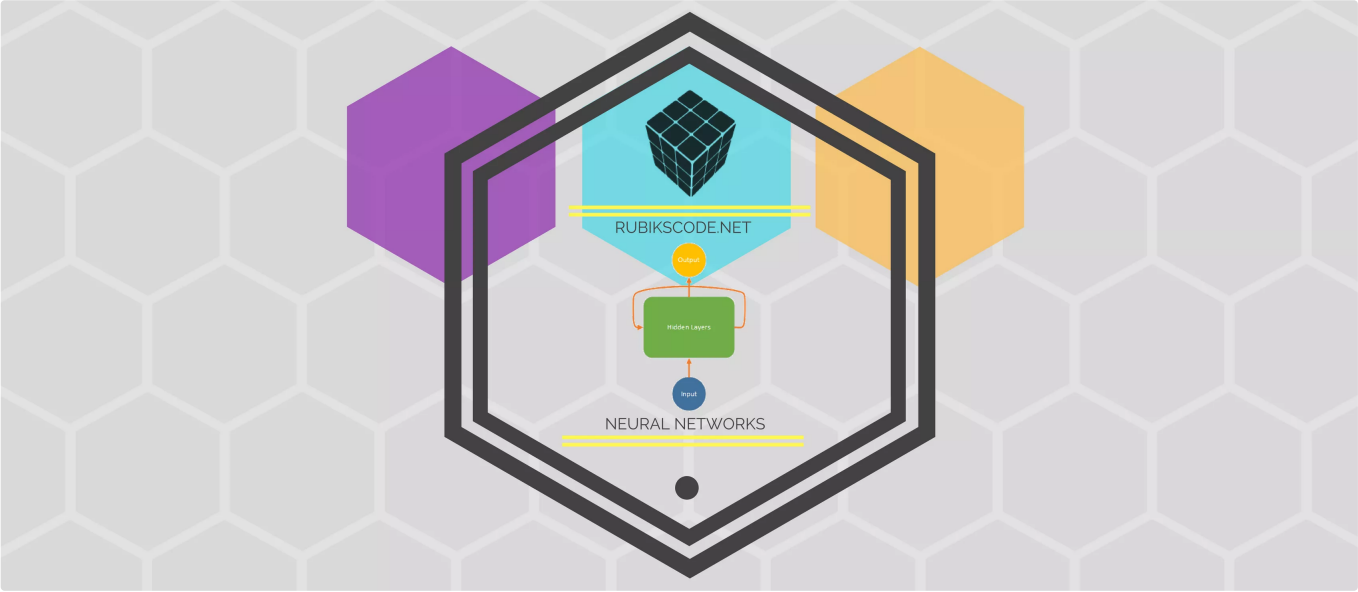
RUBIK'S CODE

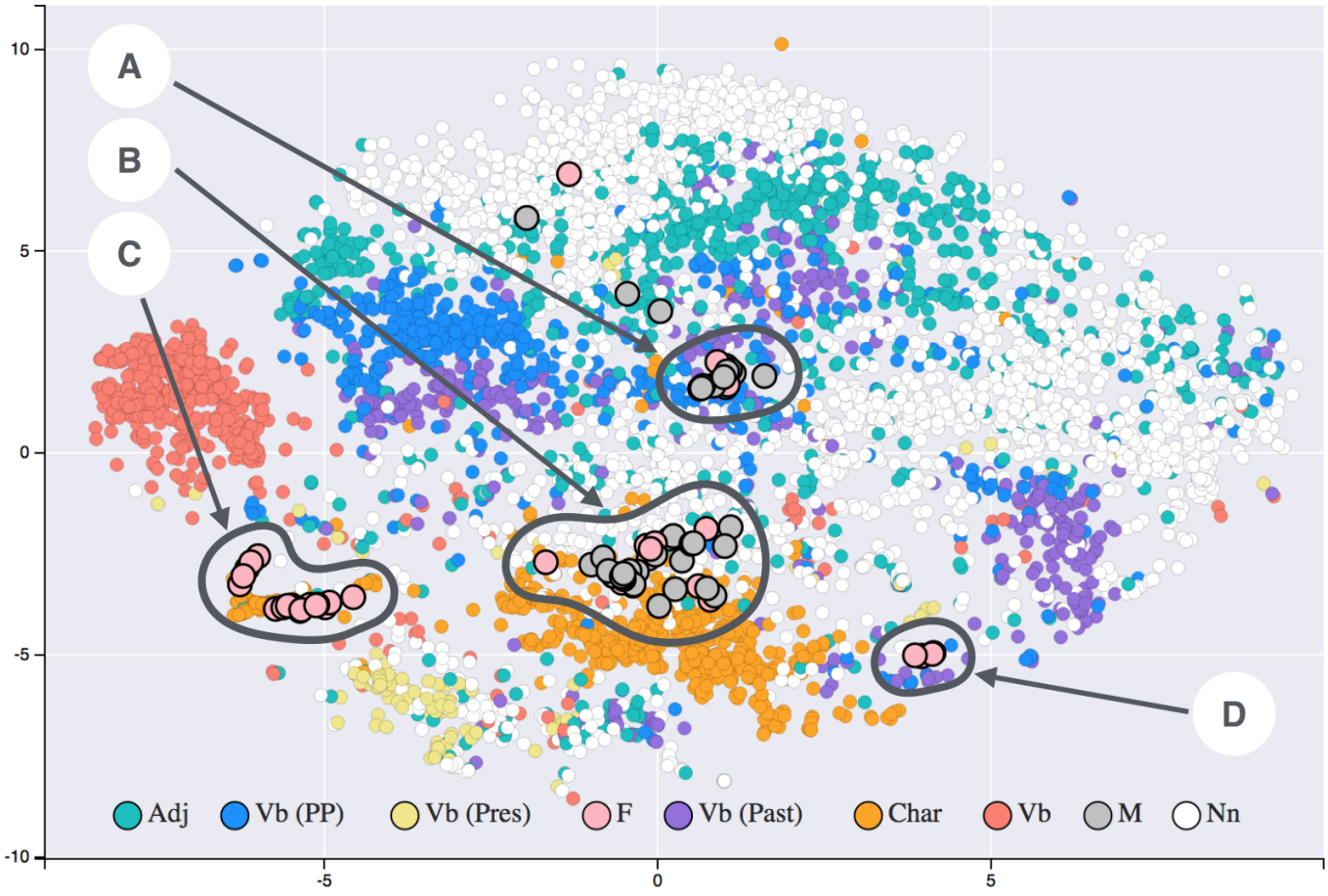
Freedom.
Wisdom.
Excellence.

Introduction to Recurrent Neural Networks

MARCH 12, 2018 — 8 COMMENTS



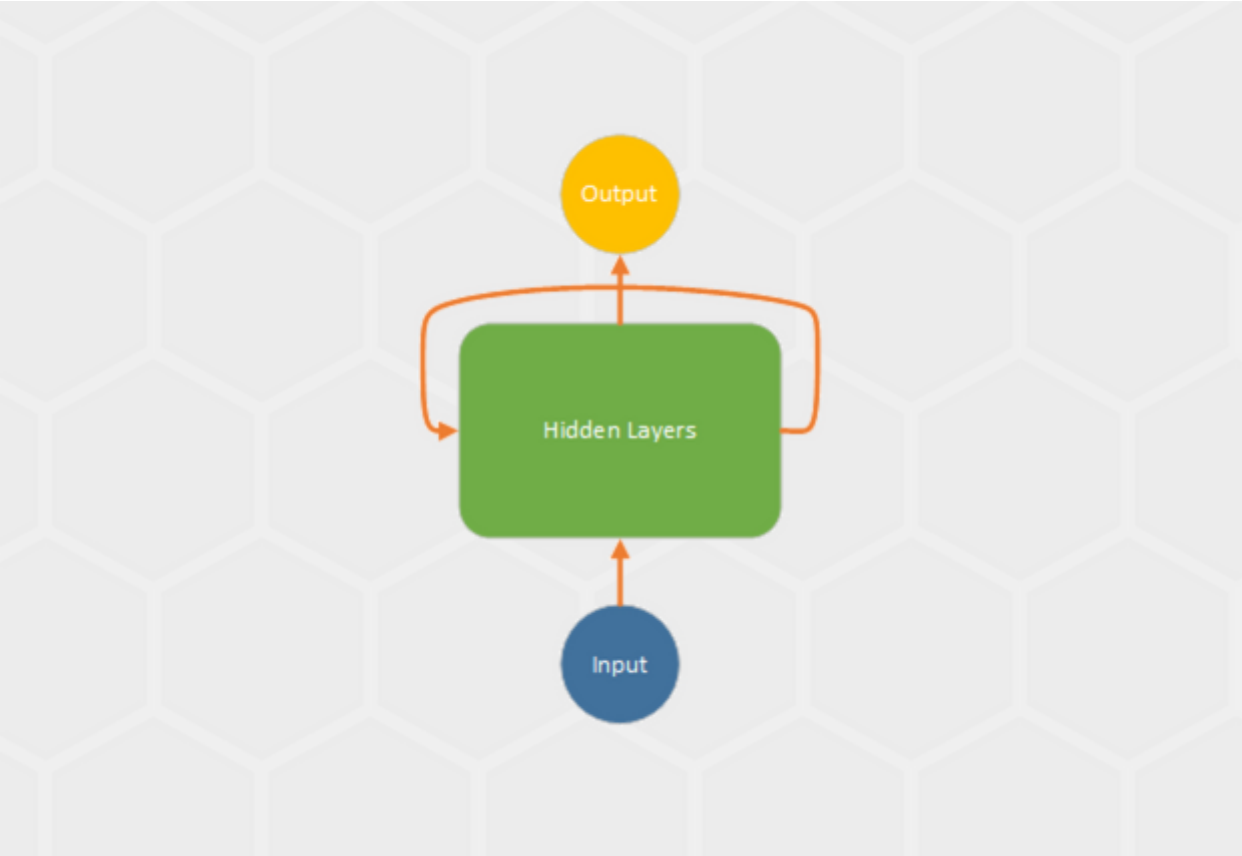
Have you ever wondered how predictive text algorithm works? How exactly does that speech recognition software know our voice? As for image classification, **convolutional neural networks** were turning the whiles behind the scene, for these kinds of problems we are using Recurrent Neural Networks (RNN). These Neural Networks are very powerful and they are especially useful in so-called Natural Language Processing (NLP). One might wonder what makes them so special.Well, the networks we examined so far, **Standard Neural Networks** and **Convolutional Neural Networks**, are accepting a fixed-size vector as input and produce a fixed-sized vector as an output.



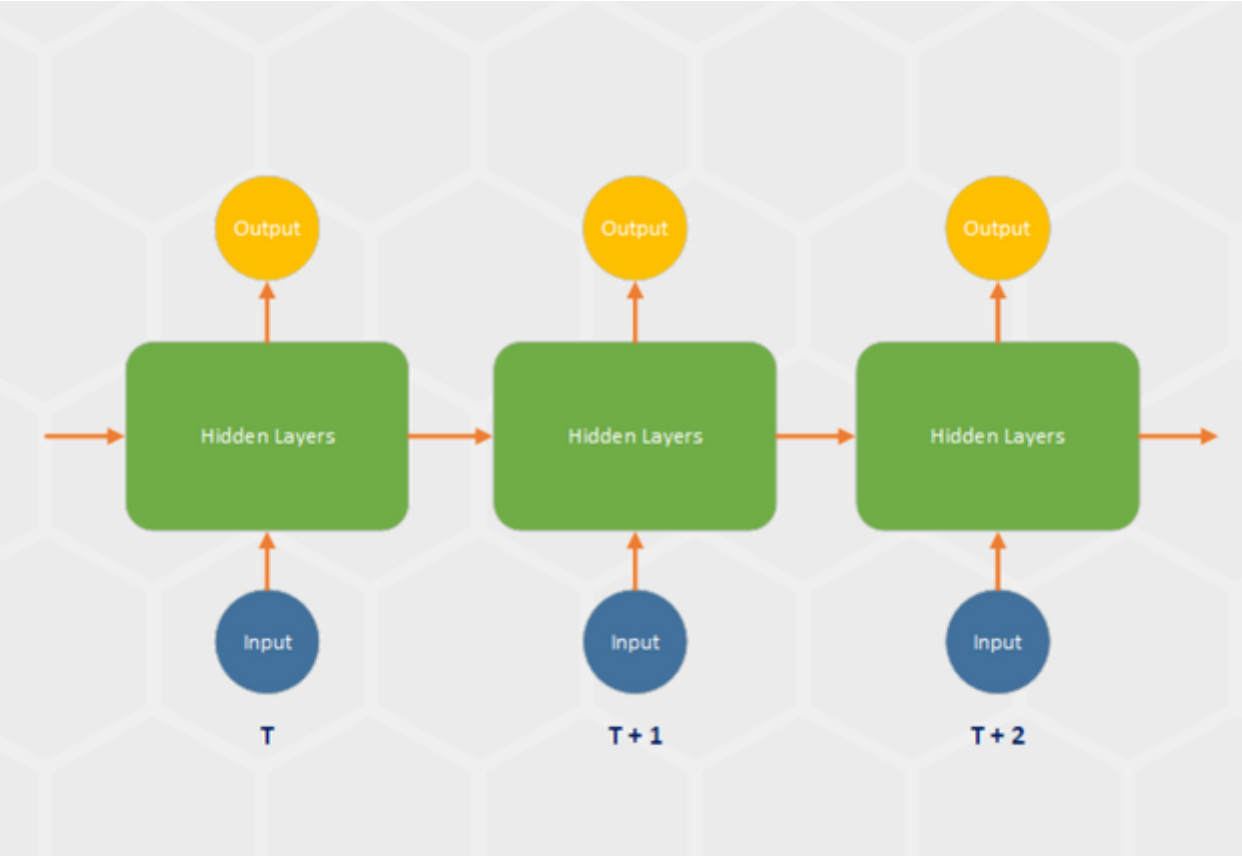
However, humans don't think like that. We are not throwing everything away and start every thought from the scratch. We use context and information we received before, as well. As you are reading this your understanding of every word is based on your understanding of previous words.This means we are thinking in sequences, ie. we are using new inputs in every nanosecond, combine them with the previous experience and form a thought based on that. That is what Recurrent Neural Networks do too (in a way), they operate over sequences of inputs and outputs and give us back the result. Using them we can make much more intelligent systems.

Architecture

The structure of Recurrent Neural Networks is the same as the structure of Artificial Neural Networks, but with one twist. They are propagating output of the network back to the input. Wait, what? Yes, we are using the output of the network in time moment *T* to calculate the output of the network in moment *T + 1.* Take a look at this oversimplified representation of RNN:

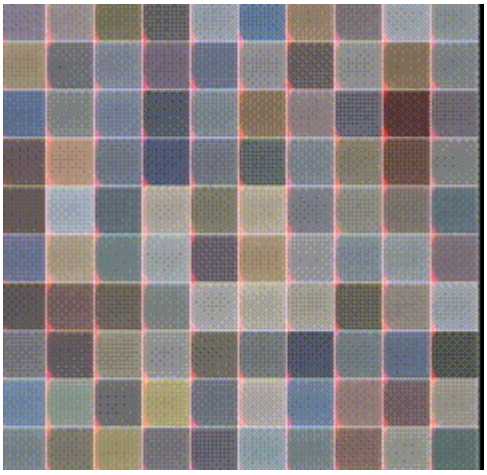


This means that output will be sent back to the input and then used when next input comes along, in next time step. Basically, “state” of the network is forward propagated through time. We can *unroll* this structure and get another way to represent the same thing is like this:



We can consider this point of view, that RNNs have “memory” in which they store information about what has been calculated so far as well. Either way, you get the picture, we are using some sort of a process to combine an output of previous time step and input in current time step to calculate output in current timestep.

One might think that sequences of inputs or outputs are rare, however, it is important to realize that we can process any data in this manner even though inputs/outputs are fixed vectors. For example, in the image below we can see how RNN generates the picture of digits by learning to sequentially add layer by layer of color to the canvas (**Gregor et al.**):



Math Behind RNNs

Recurrent Neural Networks have a simple math representation:

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

In an essence, this equation is saying that state of the network in current time step *ht* can be described as a function of the state in previous time step and input in the current time step. The function *f* is usually a nonlinearity such as **tanh or ReLU**. The state of the network in current time step *ht* becomes input value for the next time step. If we take the simplest form of RNN, the one that uses tanh as the activation function, we can represent it like this:

$$h_t = \tanh (W_{hh}h_{t-1} + W_{xh}x_t)$$

Where *Whh* are the weights of the recurrent neurons, and *Wxh* are the weights of the input neurons. In this example, we are considering just one previous time step but in general, we can observe the state of multiple time steps. This way our predictions would be more precise. The output of this network is then calculated using this current state and weights on the output.

Simplified Example

As one can see math representation of RNN is not so scary. If we transfer that knowledge into a code, we would get a simple RNN class implementation. This implementation would expose one method – *timestep,* using which we can simulate time. It would take one input *x*, which would represent the input of the network in that time step. Of course, this method would return an output *y.* Inside of the class, we would keep the information about previous inputs and network states. So, here is the simplified implementation of RNN in C#:

```
1   public class RNN
2   {
3       private Matrix<double> _state;
4       private Matrix<double> _inputWeights;
5       private Matrix<double> _recurrentWeights;
6       private Matrix<double> _outputWeights;
7
8       public RNN(Matrix<double> initialInputWeights, Matrix<double> initialReccurentWeights, Ma
9       {
```

```
10            _inputWeights = initialInputWeights;
11            _recurrentWeights = initialReccurentWeights;
12            _outputWeights = initialOutputWeights;
13        }
14
15        public Matrix<double> TimeStep(Matrix<double> input)
16        {
17            _state = Matrix<double>.Tanh(_state.Multiply(_recurrentWeights) + input.Multiply(_inp
18            return _state.Multiply(_outputWeights);
19        }
20    }
```

rnn.cs hosted with ♥ by GitHub view raw

Just one note here, for using *Matrix* type you need to install *MathNet.Numerics* NuGet package. We are using *tanh* function to squashes the activations to the range [-1, 1]. Notice that inside that function call we are multiplying current state with the recurrent weights and sum that with the multiplication of the input with the corresponding input weights. Just like in the equation from the previous chapter. An output is then calculated by multiplying that current state with output weights.

Since Python is the go-to language when it comes to implementing neural networks, here is the implementation using it as well:

```
1    import numpy as np
2
3    class RNN:
4
5      def step(self, x):
6        # Update the state
7        self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
8        # Calculate the output
9        y = np.dot(self.W_hy, self.h)
10       return y
```

rnn.py hosted with ♥ by GitHub view raw

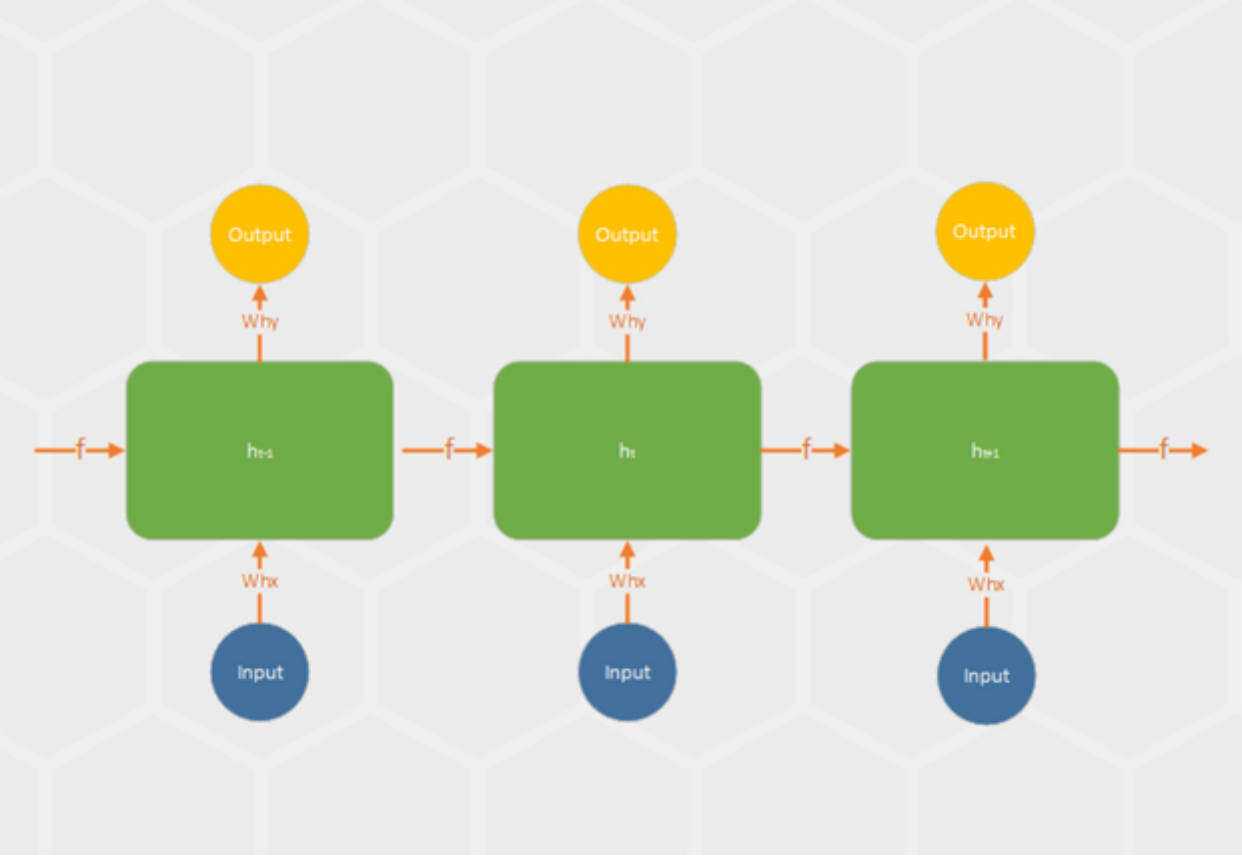
Here we used *numpy* for operations on matrices. We have done the same thing, calculated current state *h* using stored state, input *x* and weights on recurrent and input layers. We used *np.tanh* function for activation function and *np.dot* function for matrix multiplication. Once the state is calculated it is used for calculating the output.

Of course, this is a just simplified representation of the Recurrent Neural Network. The idea of this example is to give us the feeling of how the state of the networks is preserved through time.

Backpropagation Through Time (BPTT)

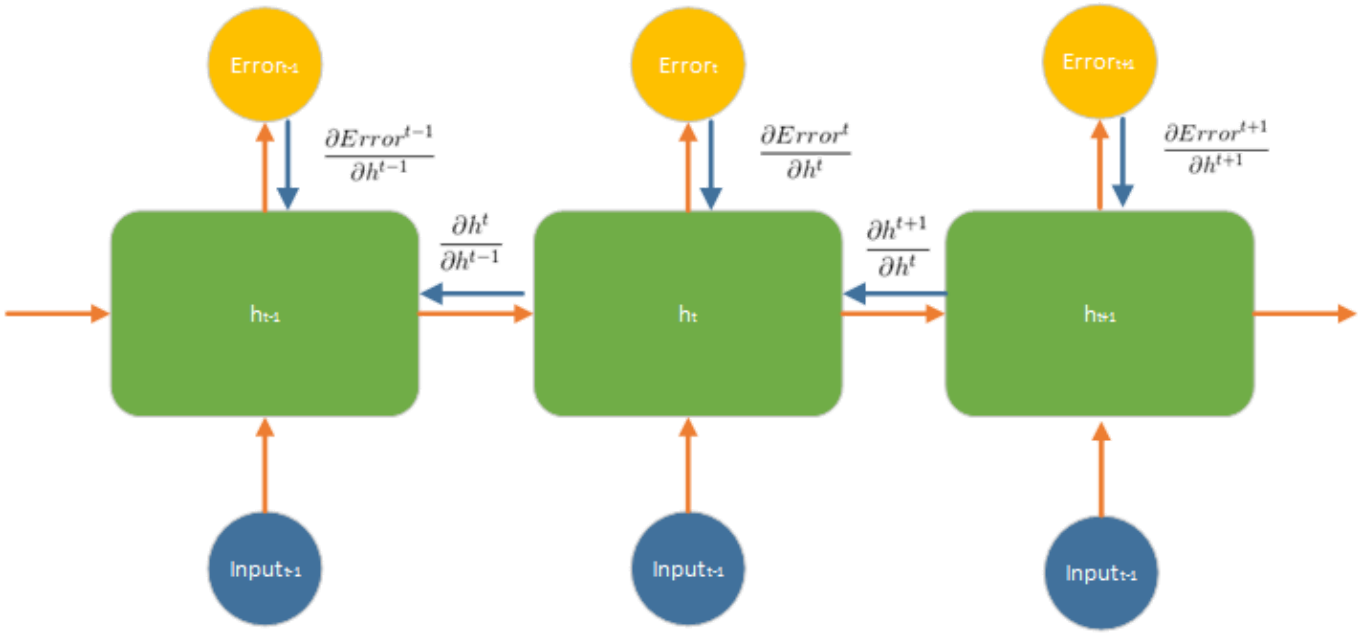
Backpropagation is a mechanism that neural networks use to update weights. In a nutshell, during the **training process** networks calculate output for some input training set of data. Then they compare that result to the desired one and according to that update weights going from output layer back to the input layer. This algorithm is a bit more complicated and you can read more about it **here**.

However, in Recurrent Neural Networks, we have an even more complicated situation. That is because we have an extra dimension – time. Figuratively speaking, we have to go back in time to update the weights. That is why this process in Recurrent Neural Networks is called Backpropagation Through Time (BPTT).



Take a look at the image above. It is the same representation of unfolded RNN, but we have added additional information from the RNN equation. The unrolled RNN kinda remind us of the standard neural network representation. That is why **backpropagation algorithm** in RNN is

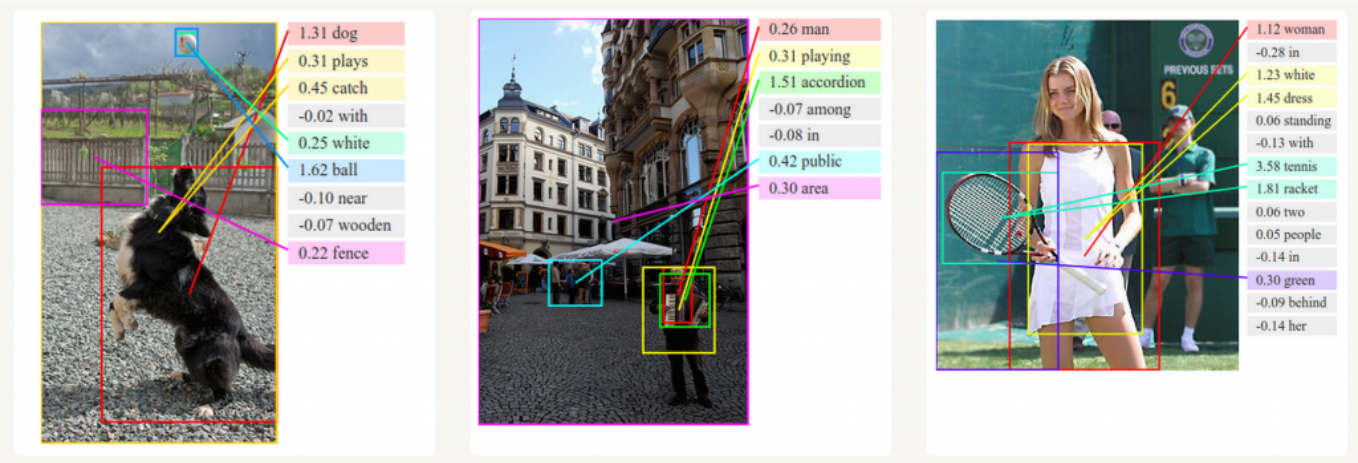
similar to the algorithm in standard neural networks. The only difference is that we summarize the gradients of the error for all time steps. This is done like this because we share parameters across layers. Here is how it is done!



Usually, the whole sequence of data is considered to be one training example. This simplifies this problem a lot because we can calculate the error in each time step and calculate the global error (as the sum of all errors). Another thing we can notice is that states are co-dependent. We can calculate gradient, using Stochastic Gradient Descent, and pass this information to the previous time step and use it for calculation of their error and gradient, and so on. This is how we squash time dimension and use regular backpropagation algorithm for aligning the weights.

Conclusion

Recurrent Neural Networks are one very useful tool with a wide range of applications. They are used in various language modeling and text generators solutions. Also, they are used for speech recognition. When combined with Convolutional Neural Networks, this kind of neural networks are used for creating labels for images that are not labeled. It is amazing how this combination works.



Source

Recurrent Neural Networks have one problem though. They are having difficulties learning long-range dependencies, meaning they don't understand interactions between data that are several steps apart. For example, sometimes we need more context when predicting words than just one previous word. This problem is called vanishing gradient problem, and it is solved by special kind of Recurrent Neural Networks – Long-Short Term Memory Networks (LSTM), the bigger topic that will be covered in the later articles.

Thanks for reading!

This article is a part of **Artificial Neural Networks Series**.

Read more posts from the author at **Rubik's Code**.
