**Autoregressive Integrated Moving Average Model**

It explicitly caters to a suite of standard structures in time series data, and as such provides a simple yet powerful method for making skillful time series forecasts.

ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration.

This acronym is descriptive, capturing the key aspects of the model itself. Briefly, they are:

* **AR**: *Autoregression*. A model that uses the dependent relationship between an observation and some number of lagged observations.
* **I**: *Integrated*. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
* **MA**: *Moving Average*. A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

Each of these components are explicitly specified in the model as a parameter. A standard notation is used of ARIMA(p,d,q) where the parameters are substituted with integer values to quickly indicate the specific ARIMA model being used.

The parameters of the ARIMA model are defined as follows:

* **p**: The number of lag observations included in the model, also called the lag order.
* **d**: The number of times that the raw observations are differenced, also called the degree of differencing.
* **q**: The size of the moving average window, also called the order of moving average.

A linear regression model is constructed including the specified number and type of terms, and the data is prepared by a degree of differencing in order to make it stationary, i.e. to remove trend and seasonal structures that negatively affect the regression model.

A value of 0 can be used for a parameter, which indicates to not use that element of the model. This way, the ARIMA model can be configured to perform the function of an ARMA model, and even a simple AR, I, or MA model.

Adopting an ARIMA model for a time series assumes that the underlying process that generated the observations is an ARIMA process. This may seem obvious, but helps to motivate the need to confirm the assumptions of the model in the raw observations and in the residual errors of forecasts from the model.

Next, let’s take a look at how we can use the ARIMA model in Python. We will start with loading a simple univariate time series.

Steps

1. Split the dataset into training and test sets.
2. Walk the time steps in the test dataset.
   1. Train an ARIMA model.
   2. Make a one-step prediction.
   3. Store prediction; get and store actual observation.
3. Calculate error score for predictions compared to expected values.

**Vector Auto Regression**

**Machine learning**

**Supervised Learning**

**Unsupervised Learning**

**Deep learning**

**Neural Networks**

**Convolution Neural Network**

## 1. What are Neural Networks

Neural networks are a type of machine learning models which are designed to operate similar to biological neurons and human nervous system. These models are used to recognize complex patterns and relationships that exists within a labelled dataset. They have following properties:

1. The core architecture of a Neural Network model is comprised of a large number of simple processing nodes called Neurons which are interconnected and organized in different layers.
2. An individual node in a layer is connected to several other nodes in the previous and the next layer. The inputs form one layer are received and processed to generate the output which is passed to the next layer.
3. The first layer of this architecture is often named as input layer which accepts the inputs, the last layer is named as the output layer which produces the output and every other layer between input and output layer is named is hidden layers.

### Key concepts in a Neural Network

#### A. Neuron:

A Neuron is a single processing unit of a Neural Network which are connected to different other neurons in the network. These connections repersents inputs and ouputs from a neuron. To each of its connections, the neuron assigns a “weight” (W) which signifies the importance the input and adds a bias (b) term.

#### B. Activation Functions

The activation functions are used to apply non-linear transformation on input to map it to output. The aim of activation functions is to predict the right class of the target variable based on the input combination of variables. Some of the popular activation functions are Relu, Sigmoid, and TanH.

#### C. Forward Propagation

Neural Network model goes through the process called forward propagation in which it passes the computed activation outputs in the forward direction.

Z = W\*X + b  
A = g(Z)

* g is the activation function
* A is the activation using the input
* W is the weight associated with the input
* B is the bias associated with the node

#### D. Error Computation:

The neural network learns by improving the values of weights and bias. The model computes the error in the predicted output in the final layer which is then used to make small adjustments the weights and bias. The adjustments are made such that the total error is minimized. Loss function measures the error in the final layer and cost function measures the total error of the network.

Loss = Actual\_Value - Predicted\_Value

Cost = Summation (Loss)

#### E. Backward Propagation:

Neural Network model undergoes the process called backpropagation in which the error is passed to backward layers so that those layers can also improve the associated values of weights and bias. It uses the algorithm called Gradient Descent in which the error is minimized and optimal values of weights and bias are obtained. This weights and bias adjustment is done by computing the derivative of error, derivative of weights, bias and subtracting them from the original values.

### Key components of Convolutional Neural Network.

**A. Convolutional layer:** In this layer, a kernel (or weight) matrix is used to extract low level features from the images. The kernel with its weights rotates over the image matrix in a sliding window fashion in order to obtained the convolved output. The kernel matrix behaves like a filter in an image extracting particular information from the original image matrix. During the colvolution process, The weights are learnt such that the loss function is minimized.

### 5.4 Create the CNN Model Architecture

In this step, create the convolutional neural network architecture with following layers:

1. Convolutional Layer with kernel size = 3\*3, 32 convolutional units, and RelU activation function
2. Convolutional Layer with kernel size = 3\*3, 64 convolutional units, and RelU activation function
3. Max Pooling Layer with pooling matrix size = 2\*2
4. Dropout Layer : A dropout layer is used for regularization and reducing the overfitting
5. Flatten Layer : A layer to convert the output in one dimentional array
6. Dense Layer : A dense layer is a fully connected layer in which every node is connected to every other node in the previous and next layers. In our network, it contains 128 neurons but this number can be changed for further experiments.
7. Another Dropout Layer for regularization
8. Final output layer : A dense layer with 10 neurons for generating the output class

In the simple neural network that we implemented in step 1, the loss function was LogLoss function and the optimizing Algorithm was Gradient Descent, In this neural network, we will use categorical\_crossentropy as this is a multi class classification as the loss function and Adadelta as the optimizing function.

## Motivation

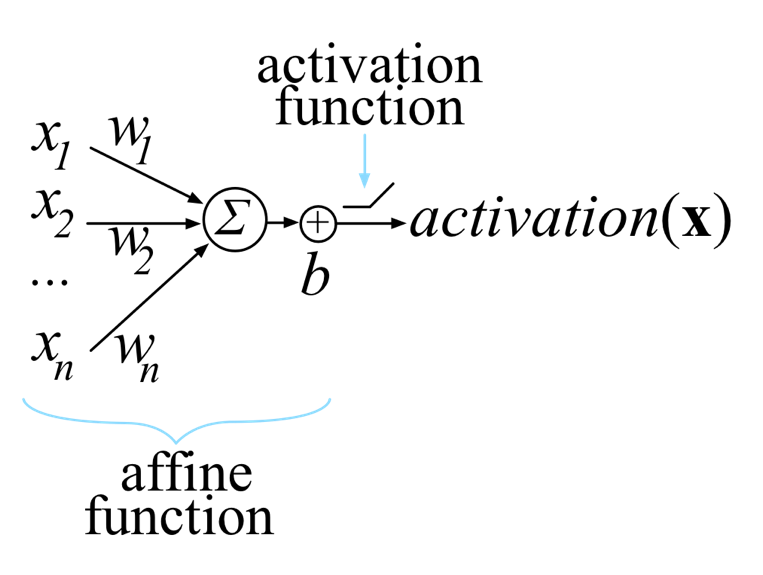
## 

## Introduction

Most of us last saw calculus in school, but derivatives are a critical part of machine learning, particularly deep neural networks, which are trained by optimizing a loss function. Pick up a machine learning paper or the documentation of a library such as [PyTorch](http://pytorch.org) and calculus comes screeching back into your life like distant relatives around the holidays. And it's not just any old scalar calculus that pops up---you need differential *matrix calculus*, the shotgun wedding of [linear algebra](https://en.wikipedia.org/wiki/Linear_algebra) and [multivariate calculus](https://en.wikipedia.org/wiki/Multivariable_calculus).

Well... maybe *need* isn't the right word; Jeremy's courses show how to become a world-class deep learning practitioner with only a minimal level of scalar calculus, thanks to leveraging the automatic differentiation built in to modern deep learning libraries. But if you really want to really understand what's going on under the hood of these libraries, and grok academic papers discussing the latest advances in model training techniques, you'll need to understand certain bits of the field of matrix calculus.

For example, the activation of a single computation unit in a neural network is typically calculated using the dot product (from linear algebra) of an edge weight vector w with an input vector x plus a scalar bias (threshold): . Function is called the unit's *affine function* and is followed by a [rectified linear unit](https://goo.gl/7BXceK), which clips negative values to zero: . Such a computational unit is sometimes referred to as an “artificial neuron” and looks like:



Neural networks consist of many of these units, organized into multiple collections of neurons called *layers*. The activation of one layer's units become the input to the next layer's units. The activation of the unit or units in the final layer is called the network output.

*Training* this neuron means choosing weights w and bias b so that we get the desired output for all N inputs x. To do that, we minimize a *loss function* that compares the network's final with the (desired output of x) for all input x vectors. To minimize the loss, we use some variation on gradient descent, such as plain [stochastic gradient descent](https://en.wikipedia.org/wiki/Stochastic_gradient_descent) (SGD), SGD with momentum, or [Adam](https://en.wikipedia.org/wiki/Stochastic_gradient_descent#Adam). All of those require the partial derivative (the gradient) of with respect to the model parameters w and b. Our goal is to gradually tweak w and b so that the overall loss function keeps getting smaller across all x inputs.

If we're careful, we can derive the gradient by differentiating the scalar version of a common loss function (mean squared error):

But this is just one neuron, and neural networks must train the weights and biases of all neurons in all layers simultaneously. Because there are multiple inputs and (potentially) multiple network outputs, we really need general rules for the derivative of a function with respect to a vector and even rules for the derivative of a vector-valued function with respect to a vector.

This article walks through the derivation of some important rules for computing partial derivatives with respect to vectors, particularly those useful for training neural networks. This field is known as *matrix calculus*, and the good news is, we only need a small subset of that field, which we introduce here. While there is a lot of online material on multivariate calculus and linear algebra, they are typically taught as two separate undergraduate courses so most material treats them in isolation. The pages that do discuss matrix calculus often are really just lists of rules with minimal explanation or are just pieces of the story. They also tend to be quite obscure to all but a narrow audience of mathematicians, thanks to their use of dense notation and minimal discussion of foundational concepts. (See the annotated list of resources at the end.)

In contrast, we're going to rederive and rediscover some key matrix calculus rules in an effort to explain them. It turns out that matrix calculus is really not that hard! There aren't dozens of new rules to learn; just a couple of key concepts. Our hope is that this short paper will get you started quickly in the world of matrix calculus as it relates to training neural networks. We're assuming you're already familiar with the basics of neural network architecture and training. If you're not, head over to [Jeremy's course](http://course.fast.ai) and complete part 1 of that, then we'll see you back here when you're done. (Note that, unlike many more academic approaches, we strongly suggest *first* learning to train and use neural networks in practice and *then* study the underlying math. The math will be much more understandable with the context in place; besides, it's not necessary to grok all this calculus to become an effective practitioner.)

*A note on notation*: Jeremy's course exclusively uses code, instead of math notation, to explain concepts since unfamiliar functions in code are easy to search for and experiment with. In this paper, we do the opposite: there is a lot of math notation because one of the goals of this paper is to help you understand the notation that you'll see in deep learning papers and books. At the [end of the paper](https://explained.ai/matrix-calculus/index.html#notation), you'll find a brief table of the notation used, including a word or phrase you can use to search for more details.

## The gradient of neuron activation

We now have all of the pieces needed to compute the derivative of a typical neuron activation for a single neural network computation unit with respect to the model parameters, w and b:

(This represents a neuron with fully connected weights and rectified linear unit activation. There are, however, other affine functions such as convolution and other activation functions, such as exponential linear units, that follow similar logic.)

Let's worry about max later and focus on computing and . (Recall that neural networks learn through optimization of their weights and biases.) We haven't discussed the derivative of the dot product yet, , but we can use the chain rule to avoid having to memorize yet another rule. (Note notation y not y as the result is a scalar not a vector.)

The dot product is just the summation of the element-wise multiplication of the elements: . (You might also find it useful to remember the linear algebra notation .) We know how to compute the partial derivatives of and but haven't looked at partial derivatives for . We need the chain rule for that and so we can introduce an intermediate vector variable u just as we did using the single-variable chain rule:

Once we've rephrased y, we recognize two subexpressions for which we already know the partial derivatives:

The vector chain rule says to multiply the partials:

To check our results, we can grind the dot product down into a pure scalar function:

Then:

Hooray! Our scalar results match the vector chain rule results.

Now, let , the full expression within the max activation function call. We have two different partials to compute, but we don't need the chain rule:

Let's tackle the partials of the neuron activation, . The use of the function call on scalar z just says to treat all negative z values as 0. The derivative of the max function is a piecewise function. When , the derivative is 0 because z is a constant. When , the derivative of the max function is just the derivative of z, which is :

An aside on broadcasting functions across scalars. When one or both of the max arguments are vectors, such as , we broadcast the single-variable function max across the elements. This is an example of an element-wise unary operator. Just to be clear:

For the derivative of the broadcast version then, we get a vector of zeros and ones where:

To get the derivative of the function, we need the chain rule because of the nested subexpression, . Following our process, let's introduce intermediate scalar variable z to represent the affine function giving:

The vector chain rule tells us:

which we can rewrite as follows:

and then substitute back in:

That equation matches our intuition. When the activation function clips affine function output z to 0, the derivative is zero with respect to any weight wi. When , it's as if the max function disappears and we get just the derivative of z with respect to the weights.

Turning now to the derivative of the neuron activation with respect to b, we get:

Let's use these partial derivatives now to handle the entire loss function.

## The gradient of the neural network loss function

Training a neuron requires that we take the derivative of our loss or “cost” function with respect to the parameters of our model, w and b. Because we train with multiple vector inputs (e.g., multiple images) and scalar targets (e.g., one classification per image), we need some more notation. Let

where , and then let

where yi is a scalar. Then the cost equation becomes:

Following our chain rule process introduces these intermediate variables:

Let's compute the gradient with respect to w first.

### The gradient with respect to the weights

From before, we know:

and

Then, for the overall gradient, we get:

To interpret that equation, we can substitute an error term yielding:

From there, notice that this computation is a weighted average across all xi in X. The weights are the error terms, the difference between the target output and the actual neuron output for each xi input. The resulting gradient will, on average, point in the direction of higher cost or loss because large ei emphasize their associated xi. Imagine we only had one input vector, , then the gradient is just . If the error is 0, then the gradient is zero and we have arrived at the minimum loss. If is some small positive difference, the gradient is a small step in the direction of . If is large, the gradient is a large step in that direction. If is negative, the gradient is reversed, meaning the highest cost is in the negative direction.

Of course, we want to reduce, not increase, the loss, which is why the [gradient descent](https://en.wikipedia.org/wiki/Gradient_descent) recurrence relation takes the negative of the gradient to update the current position (for scalar learning rate ):

Because the gradient indicates the direction of higher cost, we want to update x in the opposite direction.

### The derivative with respect to the bias

To optimize the bias, b, we also need the partial with respect to b. Here are the intermediate variables again:

We computed the partial with respect to the bias for equation previously:

For v, the partial is:

And for the partial of the cost function itself we get:

As before, we can substitute an error term:

The partial derivative is then just the average error or zero, according to the activation level. To update the neuron bias, we nudge it in the opposite direction of increased cost:

In practice, it is convenient to combine w and b into a single vector parameter rather than having to deal with two different partials: . This requires a tweak to the input vector x as well but simplifies the activation function. By tacking a 1 onto the end of x, , becomes .

This finishes off the optimization of the neural network loss function because we have the two partials necessary to perform a gradient descent.

## Summary

Hopefully you've made it all the way through to this point. You're well on your way to understanding matrix calculus! We've included a reference that s

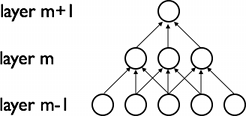
Convolutional Neural Networks (CNN) are biologically-inspired variants of MLPs. From Hubel and Wiesel’s early work on the cat’s visual cortex [[Hubel68]](http://deeplearning.net/tutorial/references.html#hubel68), we know the visual cortex contains a complex arrangement of cells. These cells are sensitive to small sub-regions of the visual field, called a receptive field. The sub-regions are tiled to cover the entire visual field. These cells act as local filters over the input space and are well-suited to exploit the strong spatially local correlation present in natural images.

Additionally, two basic cell types have been identified: Simple cells respond maximally to specific edge-like patterns within their receptive field. Complex cells have larger receptive fields and are locally invariant to the exact position of the pattern.

The animal visual cortex being the most powerful visual processing system in existence, it seems natural to emulate its behavior. Hence, many neurally-inspired models can be found in the literature. To name a few: the NeoCognitron [[Fukushima]](http://deeplearning.net/tutorial/references.html#fukushima), HMAX [[Serre07]](http://deeplearning.net/tutorial/references.html#serre07) and LeNet-5 [[LeCun98]](http://deeplearning.net/tutorial/references.html#lecun98), which will be the focus of this tutorial.

## Sparse Connectivity

CNNs exploit spatially-local correlation by enforcing a local connectivity pattern between neurons of adjacent layers. In other words, the inputs of hidden units in layer **m** are from a subset of units in layer **m-1**, units that have spatially contiguous receptive fields. We can illustrate this graphically as follows:

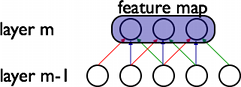


Imagine that layer **m-1** is the input retina. In the above figure, units in layer **m** have receptive fields of width 3 in the input retina and are thus only connected to 3 adjacent neurons in the retina layer. Units in layer **m+1** have a similar connectivity with the layer below. We say that their receptive field with respect to the layer below is also 3, but their receptive field with respect to the input is larger (5). Each unit is unresponsive to variations outside of its receptive field with respect to the retina. The architecture thus ensures that the learnt “filters” produce the strongest response to a spatially local input pattern.

However, as shown above, stacking many such layers leads to (non-linear) “filters” that become increasingly “global” (i.e. responsive to a larger region of pixel space). For example, the unit in hidden layer **m+1** can encode a non-linear feature of width 5 (in terms of pixel space).

## Shared Weights

In addition, in CNNs, each filter h_iis replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and form a feature map.



In the above figure, we show 3 hidden units belonging to the same feature map. Weights of the same color are shared—constrained to be identical. Gradient descent can still be used to learn such shared parameters, with only a small change to the original algorithm. The gradient of a shared weight is simply the sum of the gradients of the parameters being shared.

Replicating units in this way allows for features to be detected regardless of their position in the visual field. Additionally, weight sharing increases learning efficiency by greatly reducing the number of free parameters being learnt. The constraints on the model enable CNNs to achieve better generalization on vision problems.

## Details and Notation

A feature map is obtained by repeated application of a function across sub-regions of the entire image, in other words, by convolution of the input image with a linear filter, adding a bias term and then applying a non-linear function. If we denote the k-th feature map at a given layer as h^k, whose filters are determined by the weights W^kand bias b_k, then the feature map h^kis obtained as follows (for tanhnon-linearities):

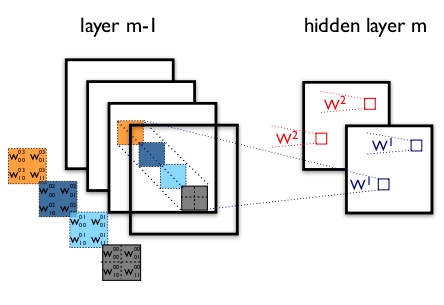
h^k_{ij} = \tanh ( (W^k * x)_{ij} + b_k ).

Note

Recall the following definition of convolution for a 1D signal. o[n] = f[n]*g[n] = \sum_{u=-\infty}^{\infty} f[u] g[n-u] = \sum_{u=-\infty}^{\infty} f[n-u] g[u].

This can be extended to 2D as follows: o[m,n] = f[m,n]*g[m,n] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f[u,v] g[m-u,n-v].

To form a richer representation of the data, each hidden layer is composed of multiple feature maps, \{h^{(k)}, k=0..K\}. The weights Wof a hidden layer can be represented in a 4D tensor containing elements for every combination of destination feature map, source feature map, source vertical position, and source horizontal position. The biases bcan be represented as a vector containing one element for every destination feature map. We illustrate this graphically as follows:



**Figure 1**: example of a convolutional layer

The figure shows two layers of a CNN. **Layer m-1** contains four feature maps. **Hidden layer m** contains two feature maps (h^0 and h^1). Pixels (neuron outputs) in h^0and h^1(outlined as blue and red squares) are computed from pixels of layer (m-1) which fall within their 2x2 receptive field in the layer below (shown as colored rectangles). Notice how the receptive field spans all four input feature maps. The weights W^0and W^1of h^0and h^1are thus 3D weight tensors. The leading dimension indexes the input feature maps, while the other two refer to the pixel coordinates.

Putting it all together, W^{kl}_{ij}denotes the weight connecting each pixel of the k-th feature map at layer m, with the pixel at coordinates (i,j) of the l-th feature map of layer (m-1).

**MaxPooling**

Another important concept of CNNs is *max-pooling,* which is a form of non-linear down-sampling. Max-pooling partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value.

Max-pooling is useful in vision for two reasons:

1. By eliminating non-maximal values, it reduces computation for upper layers.
2. It provides a form of translation invariance. Imagine cascading a max-pooling layer with a convolutional layer. There are 8 directions in which one can translate the input image by a single pixel. If max-pooling is done over a 2x2 region, 3 out of these 8 possible configurations will produce exactly the same output at the convolutional layer. For max-pooling over a 3x3 window, this jumps to 5/8.

Since it provides additional robustness to position, max-pooling is a “smart” way of reducing the dimensionality of intermediate representations.

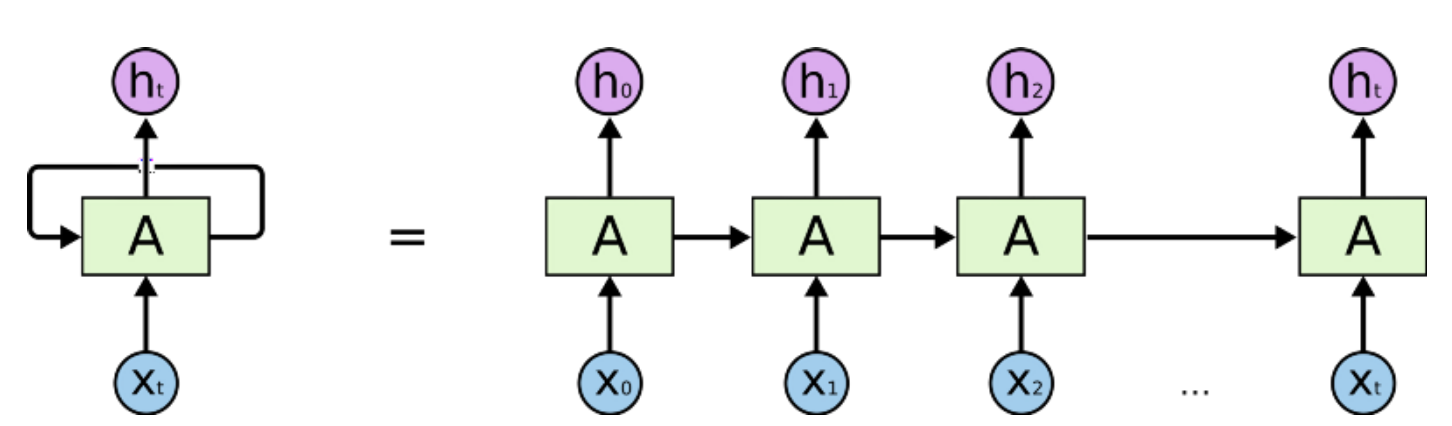
Max-pooling is done in Theano by way of theano.tensor.signal.pool.pool\_2d. This function takes as input an N dimensional tensor (where N >= 2) and a downscaling factor and performs max-pooling over the 2 trailing dimensions of the tensor.

**Recurrent Neural Network**

**LSTM**

[LSTM Neural Networks](https://en.wikipedia.org/wiki/Long_short-term_memory), which stand for **L**ong **S**hort-**T**erm **M**emory, are a particular type of recurrent neural networks that got lot of attention recently within the machine learning community.

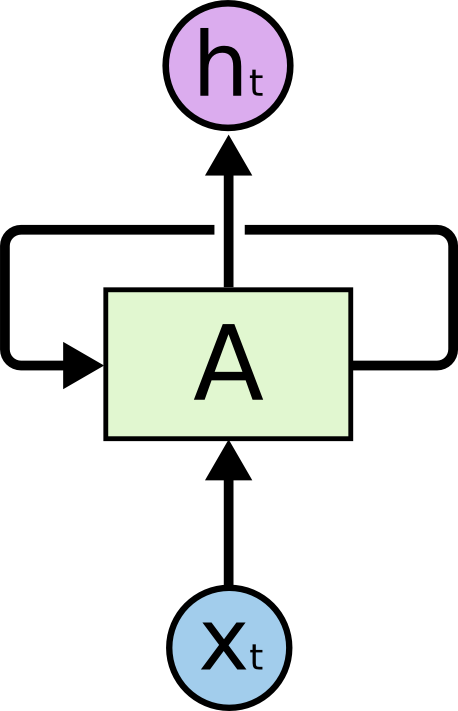
In a simple way, LSTM networks have some internal **contextual state cells**that act as long-term or short-term memory cells.  
The output of the LSTM network is **modulated** by the state of these cells. This is a very important property when we need the prediction of the neural network to depend on the **historical context** of inputs, rather than only on the very last input.



Humans don’t start their thinking from scratch every second. As you read this essay, 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. They are networks with loops in them, allowing information to persist.



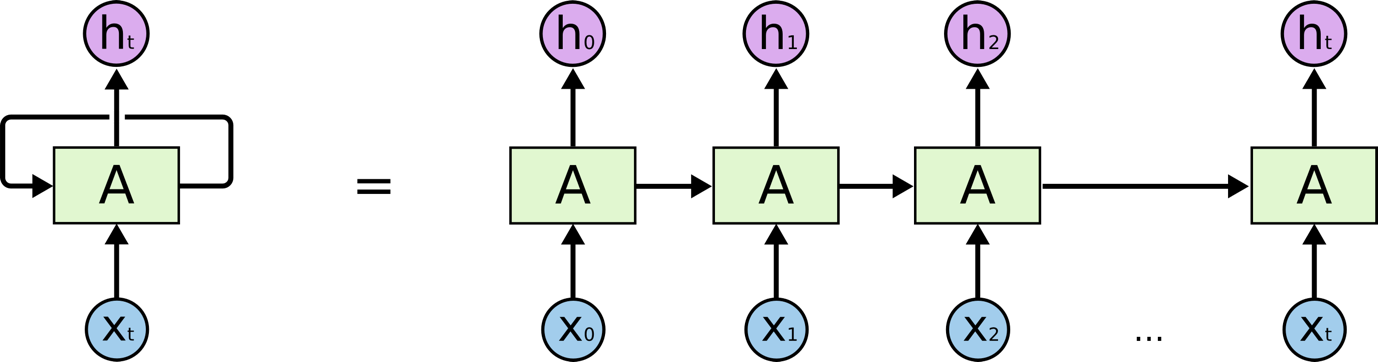
**Recurrent Neural Networks have loops.**

In the above diagram, a chunk of neural network, *A*

, looks at some input *xt* and outputs a value *ht*

. A loop allows information to be passed from one step of the network to the next.

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:



**An unrolled recurrent neural network.**

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.

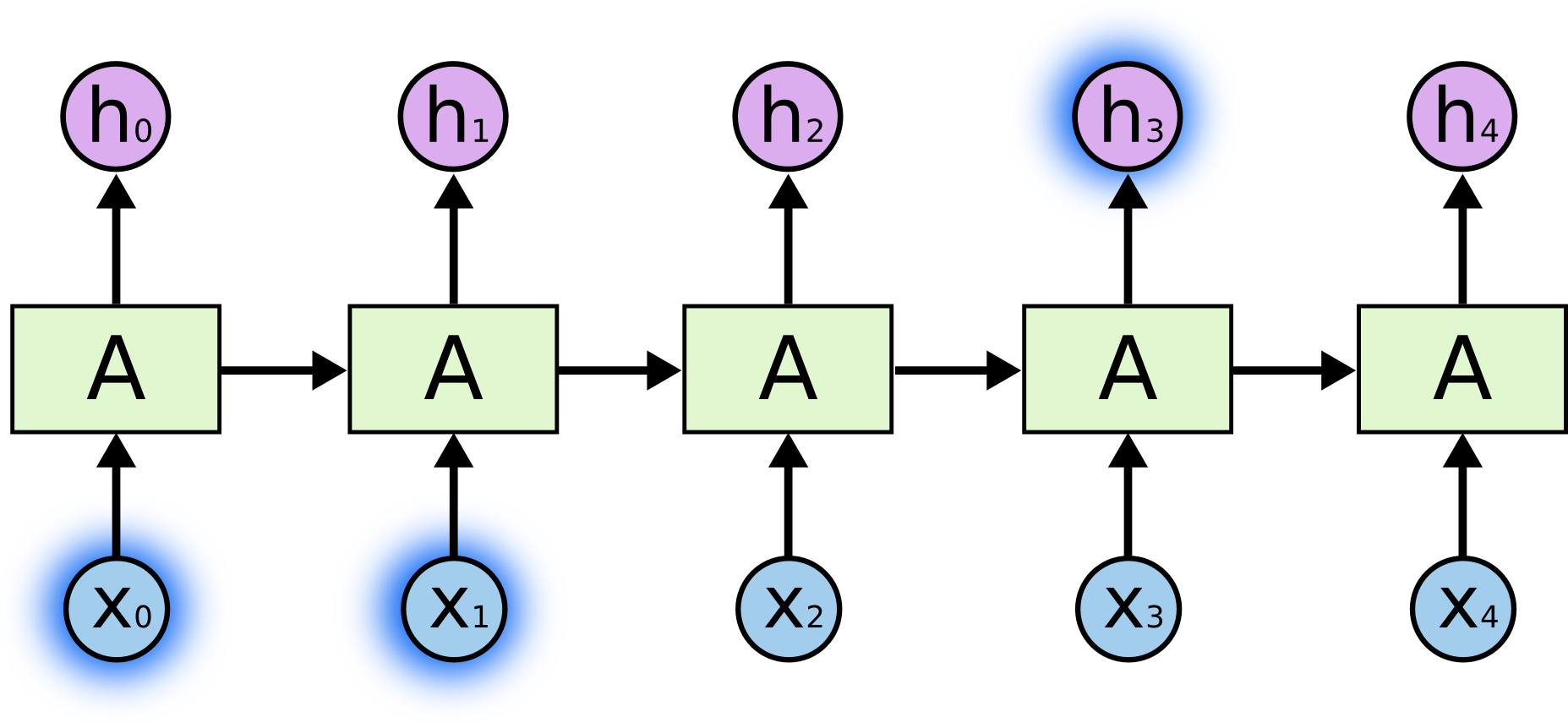
And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning… The list goes on. I’ll leave discussion of the amazing feats one can achieve with RNNs to Andrej Karpathy’s excellent blog post, [The Unreasonable Effectiveness of Recurrent Neural Networks](http://karpathy.github.io/2015/05/21/rnn-effectiveness/). But they really are pretty amazing.

Essential to these successes is the use of “LSTMs,” a very special kind of recurrent neural network which works, for many tasks, much much better than the standard version. Almost all exciting results based on recurrent neural networks are achieved with them. It’s these LSTMs that this essay will explore.

**The Problem of Long-Term Dependencies**

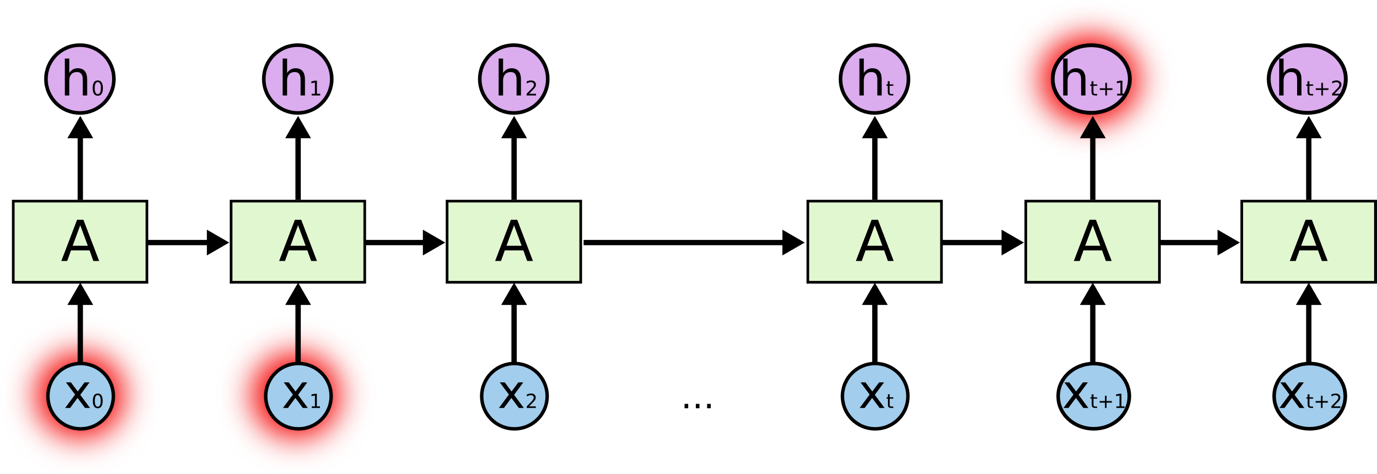
One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they’d be extremely useful. But can they? It depends.

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in “the clouds are in the *sky*,” we don’t need any further context – it’s pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it’s needed is small, RNNs can learn to use the past information.



But there are also cases where we need more context. Consider trying to predict the last word in the text “I grew up in France… I speak fluent *French*.” Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It’s entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.



In theory, RNNs are absolutely capable of handling such “long-term dependencies.” A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don’t seem to be able to learn them. The problem was explored in depth by [Hochreiter (1991) [German]](http://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf) and [Bengio, et al. (1994)](http://www-dsi.ing.unifi.it/~paolo/ps/tnn-94-gradient.pdf), who found some pretty fundamental reasons why it might be difficult.

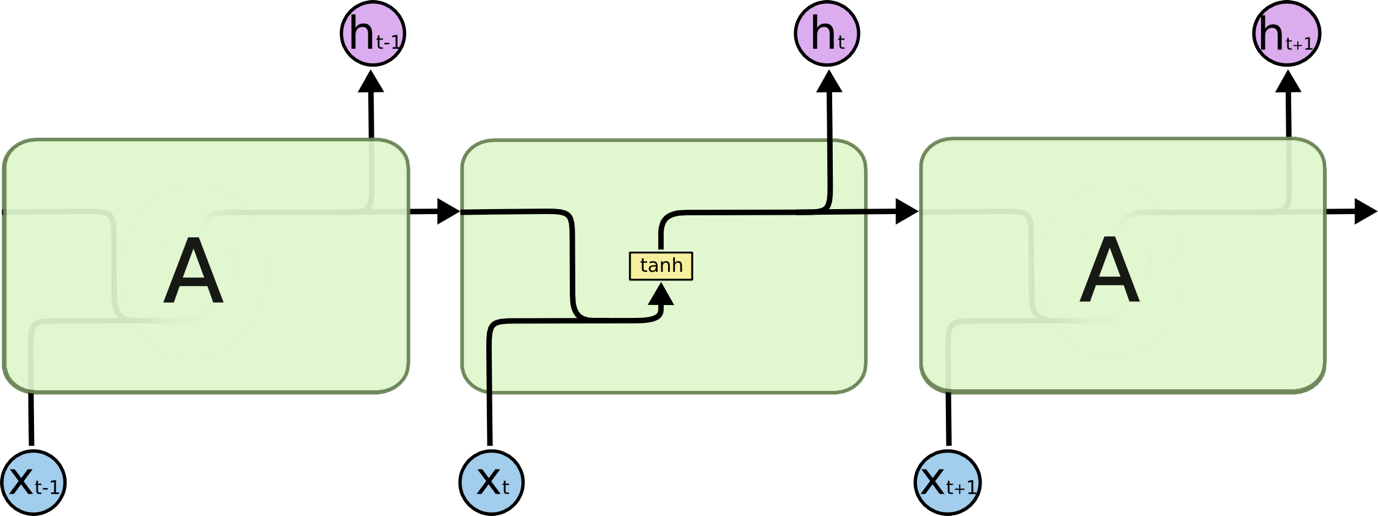
Thankfully, LSTMs don’t have this problem!

**LSTM Networks**

Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by [Hochreiter & Schmidhuber (1997)](http://www.bioinf.jku.at/publications/older/2604.pdf), and were refined and popularized by many people in following work.[1](http://colah.github.io/posts/2015-08-Understanding-LSTMs/#fn1) They work tremendously well on a large variety of problems, and are now widely used.

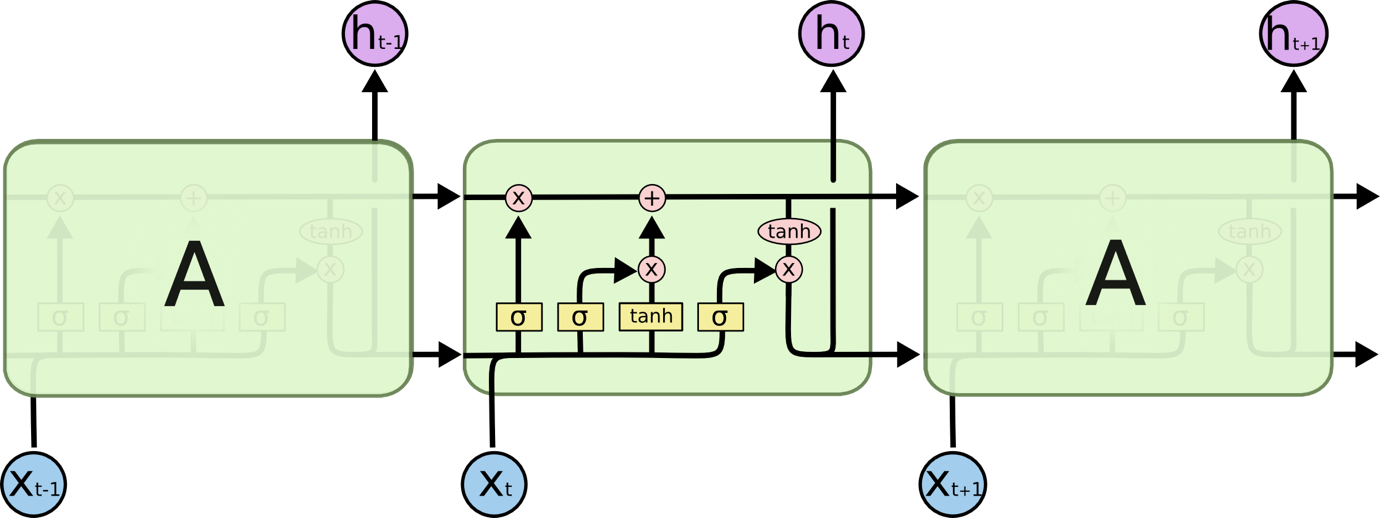
LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.



**The repeating module in a standard RNN contains a single layer.**

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



**The repeating module in an LSTM contains four interacting layers.**

Don’t worry about the details of what’s going on. We’ll walk through the LSTM diagram step by step later. For now, let’s just try to get comfortable with the notation we’ll be using.



In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

**The Core Idea Behind LSTMs**

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It’s very easy for information to just flow along it unchanged.



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”

An LSTM has three of these gates, to protect and control the cell state.

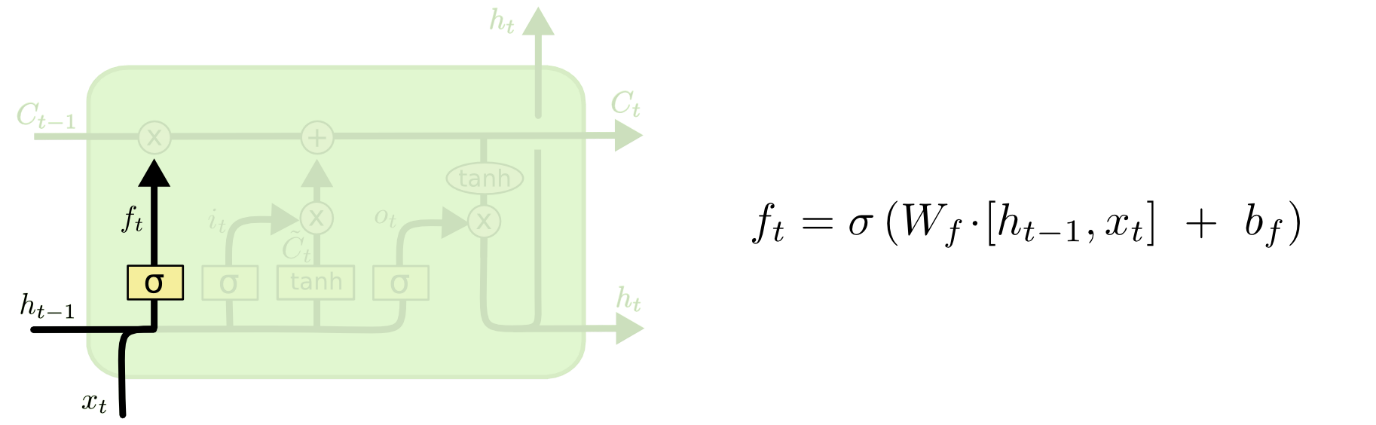
**Step-by-Step LSTM Walk Through**

The first step in our LSTM is to decide what information we’re going to throw away from the cell state. This decision is made by a sigmoid layer called the “forget gate layer.” It looks at *ht*−1

and *xt*, and outputs a number between 0 and 1 for each number in the cell state *Ct*−1. A 1 represents “completely keep this” while a 0

represents “completely get rid of this.”

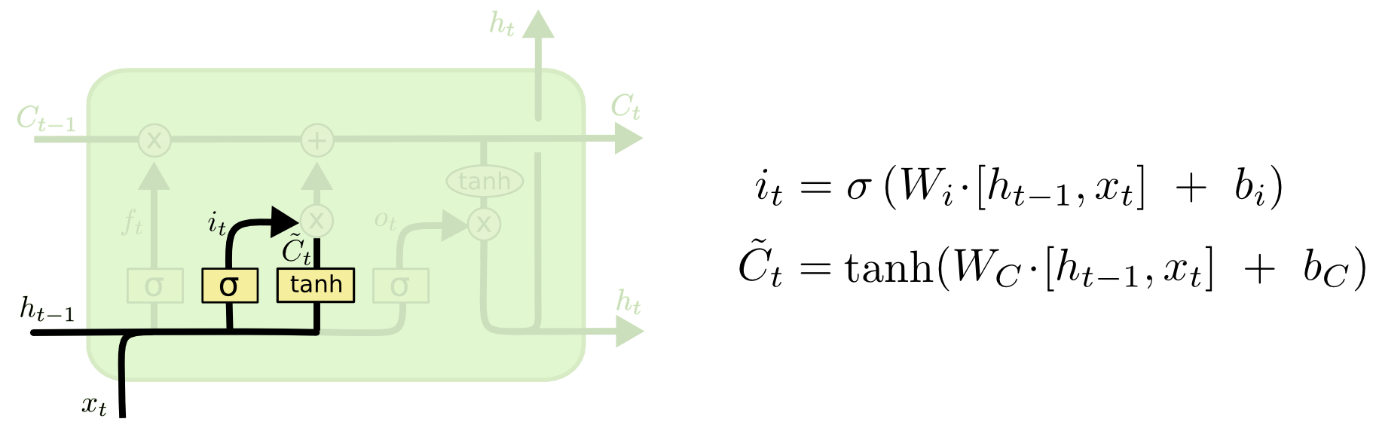
Let’s go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



The next step is to decide what new information we’re going to store in the cell state. This has two parts. First, a sigmoid layer called the “input gate layer” decides which values we’ll update. Next, a tanh layer creates a vector of new candidate values, *C*̃ *t*

, that could be added to the state. In the next step, we’ll combine these two to create an update to the state.

In the example of our language model, we’d want to add the gender of the new subject to the cell state, to replace the old one we’re forgetting.



It’s now time to update the old cell state, *Ct*−1

, into the new cell state *Ct*

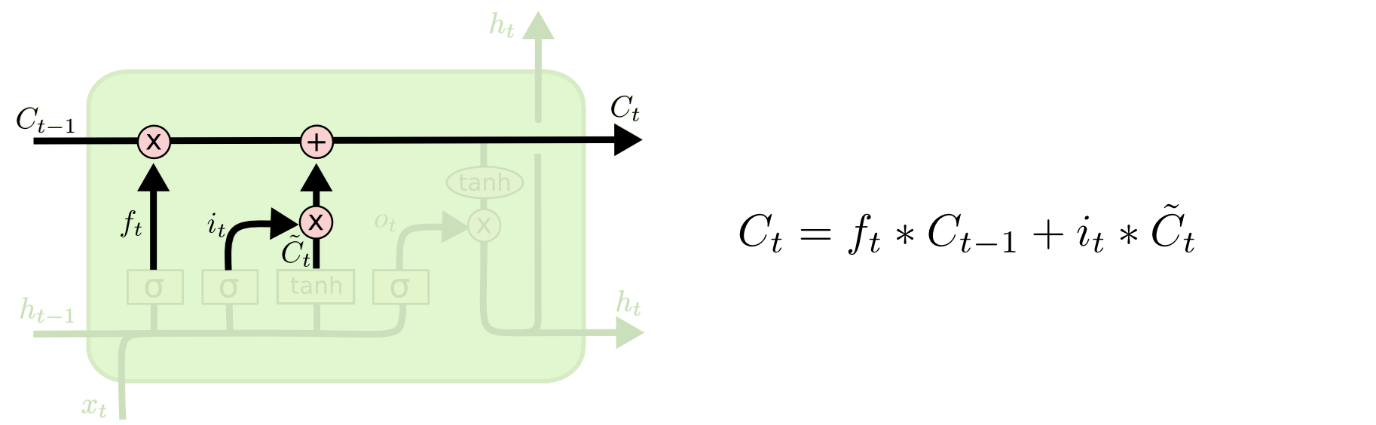
. The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by *ft*

, forgetting the things we decided to forget earlier. Then we add *it*∗*C*̃ *t*

. This is the new candidate values, scaled by how much we decided to update each state value.

In the case of the language model, this is where we’d actually drop the information about the old subject’s gender and add the new information, as we decided in the previous steps.

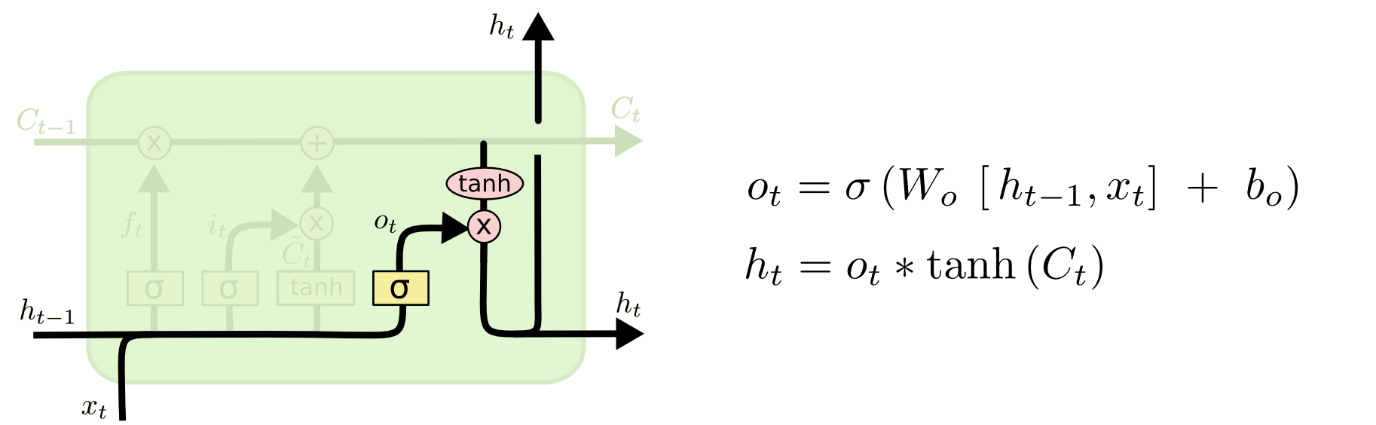


Finally, we need to decide what we’re going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we’re going to output. Then, we put the cell state through tanh

(to push the values to be between −1 and 1

) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

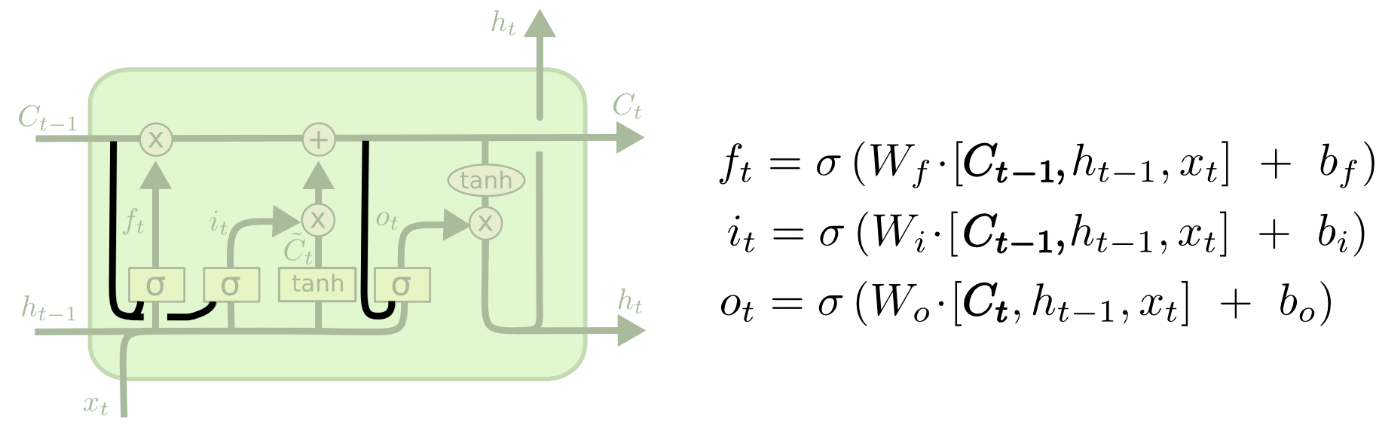
For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that’s what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that’s what follows next.



**Variants on Long Short Term Memory**

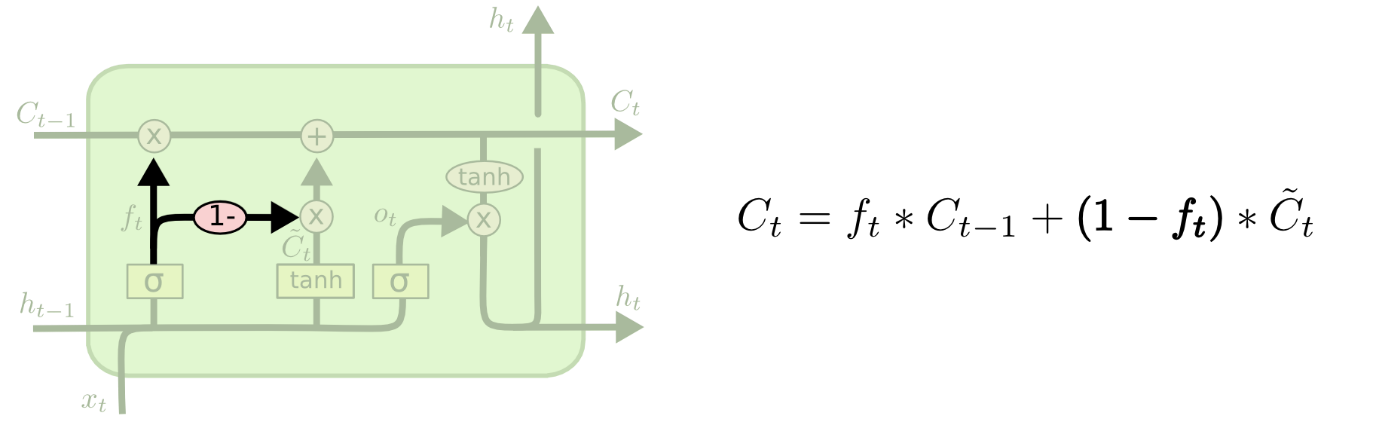
What I’ve described so far is a pretty normal LSTM. But not all LSTMs are the same as the above. In fact, it seems like almost every paper involving LSTMs uses a slightly different version. The differences are minor, but it’s worth mentioning some of them.

One popular LSTM variant, introduced by [Gers & Schmidhuber (2000)](ftp://ftp.idsia.ch/pub/juergen/TimeCount-IJCNN2000.pdf), is adding “peephole connections.” This means that we let the gate layers look at the cell state.

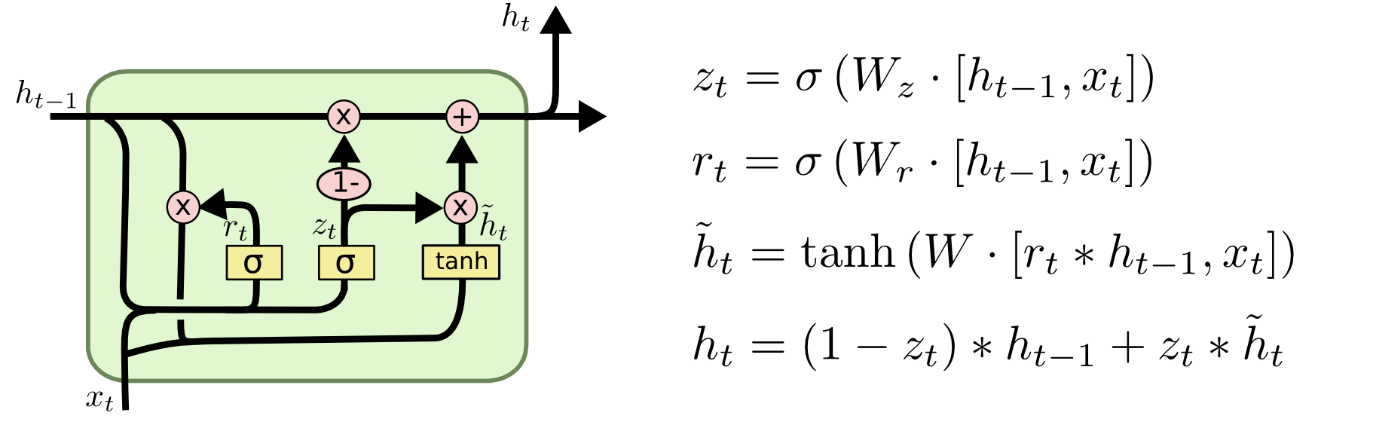


The above diagram adds peepholes to all the gates, but many papers will give some peepholes and not others.

Another variation is to use coupled forget and input gates. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we’re going to input something in its place. We only input new values to the state when we forget something older.



A slightly more dramatic variation on the LSTM is the Gated Recurrent Unit, or GRU, introduced by [Cho, et al. (2014)](http://arxiv.org/pdf/1406.1078v3.pdf). It combines the forget and input gates into a single “update gate.” It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.



These are only a few of the most notable LSTM variants. There are lots of others, like Depth Gated RNNs by [Yao, et al. (2015)](http://arxiv.org/pdf/1508.03790v2.pdf). There’s also some completely different approach to tackling long-term dependencies, like Clockwork RNNs by [Koutnik, et al. (2014)](http://arxiv.org/pdf/1402.3511v1.pdf).

Which of these variants is best? Do the differences matter? [Greff, et al. (2015)](http://arxiv.org/pdf/1503.04069.pdf) do a nice comparison of popular variants, finding that they’re all about the same. [Jozefowicz, et al. (2015)](http://jmlr.org/proceedings/papers/v37/jozefowicz15.pdf) tested more than ten thousand RNN architectures, finding some that worked better than LSTMs on certain tasks.

**Conclusion**

Earlier, I mentioned the remarkable results people are achieving with RNNs. Essentially all of these are achieved using LSTMs. They really work a lot better for most tasks!

Written down as a set of equations, LSTMs look pretty intimidating. Hopefully, walking through them step by step in this essay has made them a bit more approachable.

LSTMs were a big step in what we can accomplish with RNNs. It’s natural to wonder: is there another big step? A common opinion among researchers is: “Yes! There is a next step and it’s attention!” The idea is to let every step of an RNN pick information to look at from some larger collection of information. For example, if you are using an RNN to create a caption describing an image, it might pick a part of the image to look at for every word it outputs. In fact, [Xu, *et al.* (2015)](http://arxiv.org/pdf/1502.03044v2.pdf) do exactly this – it might be a fun starting point if you want to explore attention! There’s been a number of really exciting results using attention, and it seems like a lot more are around the corner…

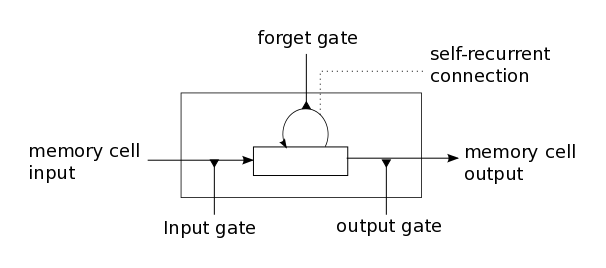
Attention isn’t the only exciting thread in RNN research. For example, Grid LSTMs by [Kalchbrenner, *et al.* (2015)](http://arxiv.org/pdf/1507.01526v1.pdf) seem extremely promising. Work using RNNs in generative models – such as [Gregor, *et al.* (2015)](http://arxiv.org/pdf/1502.04623.pdf), [Chung, *et al.* (2015)](http://arxiv.org/pdf/1506.02216v3.pdf), or [Bayer & Osendorfer (2015)](http://arxiv.org/pdf/1411.7610v3.pdf) – also seems very interesting. The last few years have been an exciting time for recurrent neural networks, and the coming ones promise to only be more so!

### LSTM

In a traditional recurrent neural network, during the gradient back-propagation phase, the gradient signal can end up being multiplied a large number of times (as many as the number of timesteps) by the weight matrix associated with the connections between the neurons of the recurrent hidden layer. This means that, the magnitude of weights in the transition matrix can have a strong impact on the learning process.

If the weights in this matrix are small (or, more formally, if the leading eigenvalue of the weight matrix is smaller than 1.0), it can lead to a situation called vanishing gradients where the gradient signal gets so small that learning either becomes very slow or stops working altogether. It can also make more difficult the task of learning long-term dependencies in the data. Conversely, if the weights in this matrix are large (or, again, more formally, if the leading eigenvalue of the weight matrix is larger than 1.0), it can lead to a situation where the gradient signal is so large that it can cause learning to diverge. This is often referred to as exploding gradients.

These issues are the main motivation behind the LSTM model which introduces a new structure called a memory cell (see Figure 1 below). A memory cell is composed of four main elements: an input gate, a neuron with a self-recurrent connection (a connection to itself), a forget gate and an output gate. The self-recurrent connection has a weight of 1.0 and ensures that, barring any outside interference, the state of a memory cell can remain constant from one timestep to another. The gates serve to modulate the interactions between the memory cell itself and its environment. The input gate can allow incoming signal to alter the state of the memory cell or block it. On the other hand, the output gate can allow the state of the memory cell to have an effect on other neurons or prevent it. Finally, the forget gate can modulate the memory cell’s self-recurrent connection, allowing the cell to remember or forget its previous state, as needed.



**Figure 1**: Illustration of an LSTM memory cell.

The equations below describe how a layer of memory cells is updated at every timestep t. In these equations:

* x_tis the input to the memory cell layer at time t
* W_i, W_f, W_c, W_o, U_i, U_f, U_c, U_oand V_oare weight matrices
* b_i, b_f, b_cand b_oare bias vectors

First, we compute the values for i_t, the input gate, and \widetilde{C_t}the candidate value for the states of the memory cells at time t:

(1)i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)

(2)\widetilde{C_t} = tanh(W_c x_t + U_c h_{t-1} + b_c)

Second, we compute the value for f_t, the activation of the memory cells’ forget gates at time t:

(3)f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)

Given the value of the input gate activation i_t, the forget gate activation f_tand the candidate state value \widetilde{C_t}, we can compute C_tthe memory cells’ new state at time t:

(4)C_t = i_t * \widetilde{C_t} + f_t * C_{t-1}

With the new state of the memory cells, we can compute the value of their output gates and, subsequently, their outputs:

(5)o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)

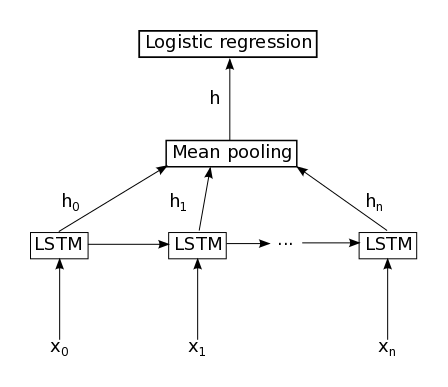
(6)h_t = o_t * tanh(C_t)

### Our model

The model we used in this tutorial is a variation of the standard LSTM model. In this variant, the activation of a cell’s output gate does not depend on the memory cell’s state C_t. This allows us to perform part of the computation more efficiently (see the implementation note, below, for details). This means that, in the variant we have implemented, there is no matrix V_oand equation [(5)](http://deeplearning.net/tutorial/lstm.html#equation-5) is replaced by equation [(7)](http://deeplearning.net/tutorial/lstm.html#equation-5-alt):

(7)o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)

Our model is composed of a single LSTM layer followed by an average pooling and a logistic regression layer as illustrated in Figure 2 below. Thus, from an input sequence x_0, x_1, x_2, ..., x_n, the memory cells in the LSTM layer will produce a representation sequence h_0, h_1, h_2, ...,
h_n. This representation sequence is then averaged over all timesteps resulting in representation h. Finally, this representation is fed to a logistic regression layer whose target is the class label associated with the input sequence.



**Figure 2** : Illustration of the model used in this tutorial. It is composed of a single LSTM layer followed by mean pooling over time and logistic regression.

**Implementation note** : In the code included this tutorial, the equations [(1)](http://deeplearning.net/tutorial/lstm.html#equation-1), [(2)](http://deeplearning.net/tutorial/lstm.html#equation-2), [(3)](http://deeplearning.net/tutorial/lstm.html#equation-3) and [(7)](http://deeplearning.net/tutorial/lstm.html#equation-5-alt) are performed in parallel to make the computation more efficient. This is possible because none of these equations rely on a result produced by the other ones. It is achieved by concatenating the four matrices W_*into a single weight matrix Wand performing the same concatenation on the weight matrices U_*to produce the matrix Uand the bias vectors b_*to produce the vector b. Then, the pre-nonlinearity activations can be computed with:

z = W x_t + U h_{t-1} + b

The result is then sliced to obtain the pre-nonlinearity activations for i, f, \widetilde{C_t}, and oand the non-linearities are then applied independently for each.