Capstone Project: Stock Price Prediction using Deep learning - Neural Network (LSTM)

Surjit Singh Jagpal Singh Walia

NJCU School of Business

Author Note

This paper is being submitted on May 10, 2017, for MBA (Finance)

FINC 779 - Capstone project in Finance

Prof. Christopher Versace

Correspondence regarding this paper should be addressed to

Surjit Singh Jagpal Singh Walia, 12 Bond Street, #1, Jersey City, NJ 07306

E-mail: [swalia@njcu.edu](mailto:swalia@njcu.edu)

Telephone: 201-375-3817.

# ACKNOWLEDGEMENTS

The following individual and institutions, in no particular order, have been tremendously supportive throughout the duration of the project. We simply cannot thank them enough.

# Table of Contents

[ACKNOWLEDGEMENTS 2](#_Toc481990825)

[Table of Contents 3](#_Toc481990826)

[Abstract 4](#_Toc481990827)

[Introduction 5](#_Toc481990828)

[Prevailing Theories 5](#_Toc481990829)

[EMH Theories 5](#_Toc481990830)

[Choas Theories 5](#_Toc481990831)

[What is Neural Network (NARX)? 6](#_Toc481990832)

[What Is Deep Learning? 8](#_Toc481990833)

[Supervised learning: 9](#_Toc481990834)

[Unsupervised learning: 9](#_Toc481990835)

[Why is Deep Learning Important? 10](#_Toc481990836)

[Classification of deep learning models 11](#_Toc481990837)

[Perceptrons 13](#_Toc481990838)

[Sigmoid neurons 21](#_Toc481990839)

[The architecture of neural networks 27](#_Toc481990840)

[Understanding LSTM Networks 31](#_Toc481990841)

[The Problem of Long-Term Dependencies 33](#_Toc481990842)

[LSTM Networks 35](#_Toc481990843)

[Step-by-Step LSTM Walk Through 38](#_Toc481990844)

[Variants on Long Short Term Memory 41](#_Toc481990845)

[References 43](#_Toc481990846)

# Abstract

Deep learning is impacting everything from healthcare to transportation to manufacturing, and more. Companies are turning to deep learning to solve hard problems, like speech recognition, object recognition, and machine translation.

Forecasts of stocks prices is an act of determining the future value of company’s stock to gain potential profits. The difficulties of forecasts of stocks prices are fluctuations in human behavior, unknown company developments, and sudden micro economic turmoil.

This project is to provide forecasts of stocks prices using Deep Learning methods, such as recurrent neural networks (RNN). Stock market prediction is the act of trying to determine the future value of a company stock or other financial instrument traded on an exchange. The successful prediction of a stock's future price could yield significant profit. RNN is a class of artificial neural network where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behavior. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. This makes them applicable to tasks such as unsegmented connected Stock price prediction handwriting recognition or speech recognition. Application of artificial neural networks to the prediction of stock prices and their trends is covered in multiple academic papers. However, prediction of stock prices using deep networks requires a lot of computing power and has numerous complications and thus was not feasible until latest developments in parallel computing and big data areas.

# Introduction

Stock market prediction is the act of trying to determine the future value of a company stock or other financial instrument traded on an exchange. The successful prediction of a stock's future price could yield significant profit. So, stock price prediction is one of the most widely studied and challenging problems, attracting researchers from many fields including economics, history, finance, mathematics, and computer science. The efficient-market hypothesis suggests that stock prices reflect all currently available information and any price changes that are not based on newly revealed information thus are inherently unpredictable. Others disagree and those with this viewpoint possess myriad methods and technologies which purportedly allow them to gain future price information.

# Prevailing Theories

There are two theories prevailing for forecasts of stocks prices as list below:

EMH Theories: The efficient market hypothesis (EMH) is an investment theory that states it is impossible to "beat the market" because stock market efficiency causes existing share prices to always incorporate and reflect all relevant information. Uses linear model.

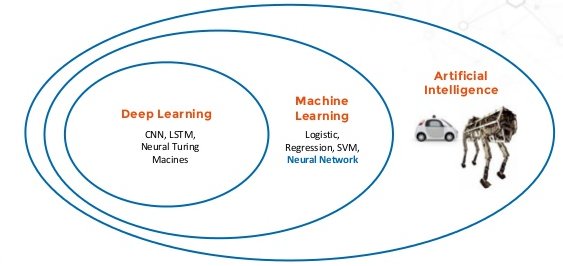
Choas Theories: is a branch of mathematics focused on the behavior of dynamical systems that are highly sensitive to initial conditions. 'Chaos' is an interdisciplinary theory stating that within the apparent randomness of chaotic complex systems, there are underlying patterns, constant feedback loops, repetition, self-similarity, fractals, self-organization, and reliance on programming at the initial point known as sensitive dependence on initial conditions. Uses dynamic models such as NARX (nonlinear autoregressive network with exogenous inputs – both historical price of stock and market sentiments).

A time series is a sequence of data points, typically measured at uniform time intervals. Examples occur in a variety of fields ranging from economics to engineering, and methods of analyzing time series constitute an important part of Statistics. Time series analysis comprises methods for analyzing time series data in order to extract meaningful characteristics of the data and forecast future values. Stock history price are time series data. The volatile nature of the stock market makes it difficult to apply simple time-series or regression techniques. A time series is a sequence of data points, typically measured at uniform time intervals. The time series analysis and statistical modeling can help you understand the trends of stock market, and is be very helpful when it comes to extract meaningful characteristics of the data and forecast future values. It's more like, you can understand the trends / patterns looking backwards. The Stock prices are made of deterministic plus random patterns that can be “learnt”.

# What is Neural Network (NARX)?

To understand what deep learning is, we first need to understand the relationship deep learning has with machine learning, neural networks, and artificial intelligence.

We have a lot of confusion around artificial intelligence, machine learning, and deep learning. Actually those terms are just a subset of the Artificial Intelligence. The best way to think of this relationship is to visualize them as concentric circles.



At the outer most ring you have artificial intelligence (using computers to reason). One layer inside of that is machine learning. With artificial neural networks and deep learning at the center.

Broadly speaking, deep learning is a more approachable name for an artificial neural network. The “deep” in deep learning refers to the depth of the network. An artificial neural network can be very shallow.

Neural networks are inspired by the structure of the cerebral cortex. At the basic level is the perceptron, the mathematical representation of a biological neuron. Like in the cerebral cortex, there can be several layers of interconnected perceptrons.

The first layer is the input layer. Each node in this layer takes an input, and then passes its output as the input to each node in the next layer. There are generally no connections between nodes in the same layer and the last layer produces the outputs.

We call the middle part the hidden layer. These neurons have no connection to the outside (e.g. input or output) and are only activated by nodes in the previous layer

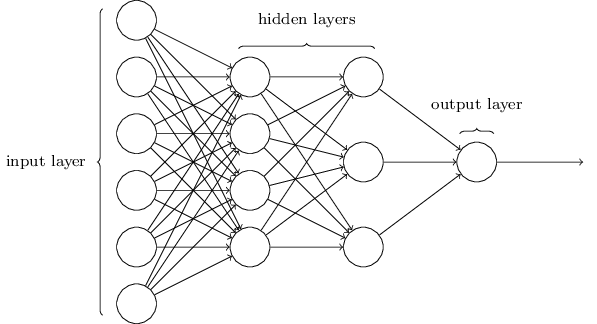


Figure : Credit: Michael A. Nielsen, “Neural Networks and Deep Learning”

# What Is Deep Learning?

Think of deep learning as the technique for learning in neural networks that utilizes multiple layers of abstraction to solve pattern recognition problems. In the 1980s, most neural networks were a single layer due to the cost of computation and availability of data.

Machine learning is considered a branch or approach of Artificial intelligence, whereas deep learning is a specialized type of machine learning.

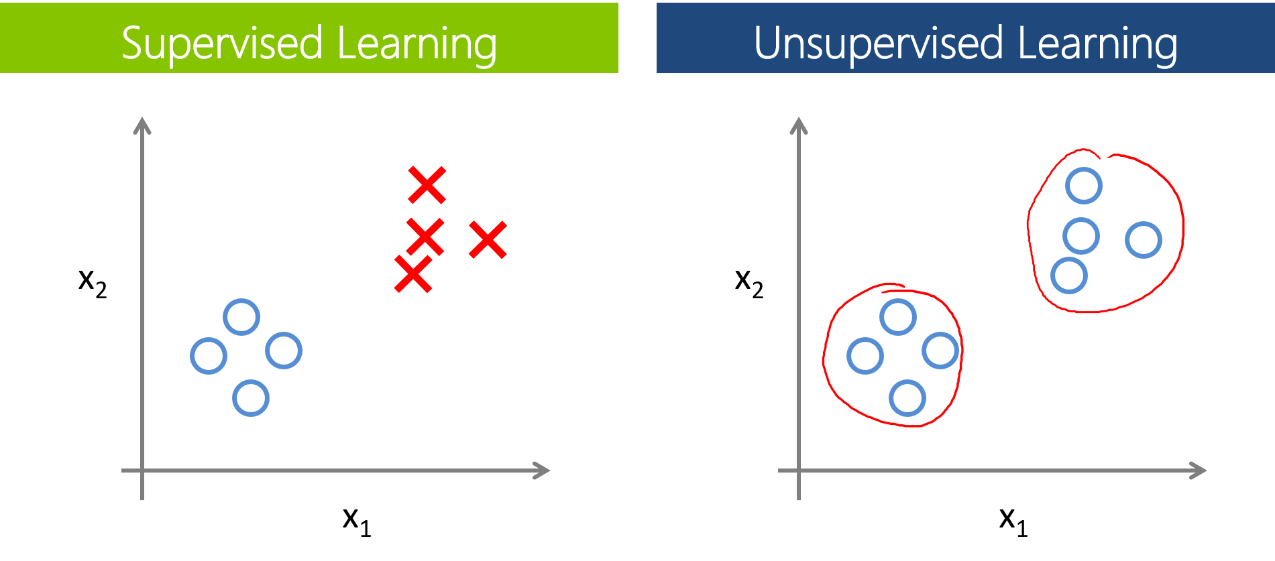
Machine learning involves computer intelligence that doesn’t know the answers up front. Instead, the program will run against training data, verify the success of its attempts, and modify its approach accordingly. Machine learning typical requires a sophisticated education, spanning software engineering and computer science to statistical methods and linear algebra.

The basic Idea of Machine Learning is to make the computer learn something from the data. There are two broad classes of machine learning methods:

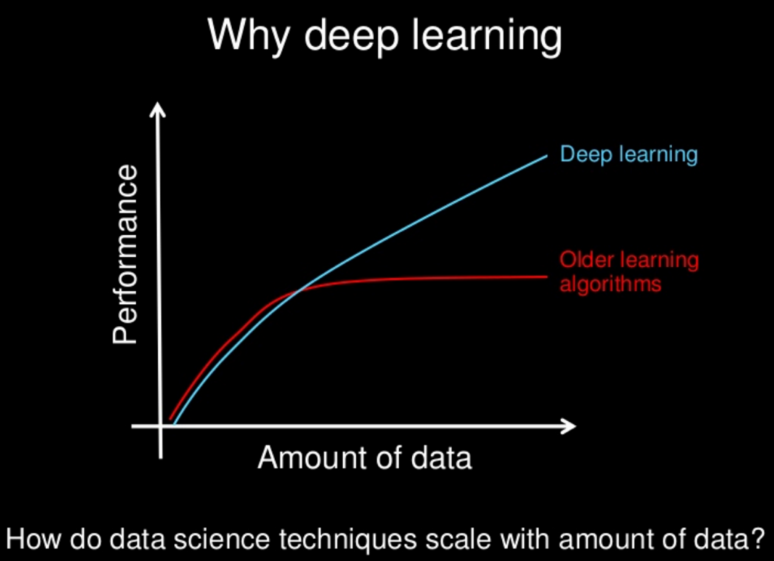
Supervised learning: In supervised learning, a machine learning algorithm uses a labeled dataset to infer the desired outcome. This takes a lot of data and time, since the data needs to be labeled by hand. Supervised learning is great for classification and regression problems.

For example, let’s say that we were running a company and want to determine the effect of bonuses on employee retention. If we had historical data – i.e. employee bonus amount and tenure – we could use supervised machine learning.

Unsupervised learning: With unsupervised learning, there aren’t any predefined or corresponding answers. The goal is to figure out the hidden patterns in the data. It’s usually used for clustering and associative tasks, like grouping customers by behavior. Amazon’s “customers who also bought…” recommendations are a type of associative task.

While supervised learning can be useful, we often have to resort to unsupervised learning. Deep learning has proven to be an effective unsupervised learning technique

# Why is Deep Learning Important?



Computers have long had techniques for recognizing features inside of images. The results weren’t always great. Computer vision has been a main beneficiary of deep learning. Computer vision using deep learning now rivals humans on many image recognition tasks.

Facebook has had great success with identifying faces in photographs by using deep learning. It’s not just a marginal improvement, but a game changer: “Asked whether two unfamiliar photos of faces show the same person, a human being will get it right 97.53 percent of the time. New software developed by researchers at Facebook can score 97.25 percent on the same challenge, regardless of variations in lighting or whether the person in the picture is directly facing the camera.”

Speech recognition is an area that’s felt deep learning’s impact. Spoken languages are so vast and ambiguous. Baidu – one of the leading search engines of China – has developed a voice recognition system that is faster and more accurate than humans at producing text on a mobile phone. In both English and Mandarin.

What is particularly fascinating, is that generalizing the two languages didn’t require much additional design effort: “Historically, people viewed Chinese and English as two vastly different languages, and so there was a need to design very different features,” Andrew Ng says, chief scientist at Baidu. “The learning algorithms are now so general that you can just learn.”

Google is now using deep learning to manage the energy at the company’s data centers. They’ve cut their energy needs for cooling by 40%. That translates to about a 15% improvement in power usage efficiency for the company and hundreds of millions of dollars in savings.

# Classification of deep learning models

All classification tasks depend upon labeled datasets; that is, humans must transfer their knowledge to the dataset in order for a neural to learn the correlation between labels and data. This is known as supervised learning.

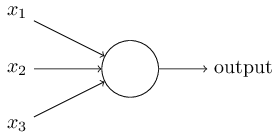
* Detect faces, identify people in images, recognize facial expressions (angry, joyful)
* Identify objects in images (stop signs, pedestrians, lane markers…)
* Recognize gestures in video
* Detect voices, identify speakers, transcribe speech to text, recognize sentiment in voices
* Classify text as spam (in emails), or fraudulent (in insurance claims); recognize sentiment in text (customer feedback)
* Any labels that humans can generate, any outcomes you care about and which correlate to data, can be used to train a neural network.

The following table attempts to show the neural nets most useful for different problems.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data Sector** | **Use Case** | **Input** | **Transform** | **Neural Net** |
| Text | * Sentiment Analysis * Named-entity recognition * Part-of-speech tagging * Semantic-role labeling | Word Vector | Gaussian Rectified | RNTN or DBN (with moving window) |
| Document | Topic Modeling/ Sematic hashing (unsupervised) | Word count probability | Can be binary | Deep Autoencoder (wrapping a DBN or SDA) |
|  | Document classification | RF-IDF (or word count prob.) | Binary | Deep-belief network, stacked denoising Autoencoder |
| Image | Image Recognition | Binary | Binary (visible and hidden) | Deep-belief network |
|  |  | Continuous | Gaussian Rectified | Deep-belief network |
|  | Multi-object recognition |  |  | Convolutional Net, RNTN (image vectorization forthcoming) |
|  | Image search / semantic hashing |  | Gaussian Rectified | Deep Autoencoder (wrapping a DBN) |
| Sound | Voice recognition |  | Gaussian Rectified | Recurrent Net |
|  |  |  |  | Moving window for DBN or ConvNet |
| Time Series | Predictive Analytics |  | Gaussian Rectified | Recurrent Net |
|  |  |  |  | Moving window for DBN or ConvNet |

A powerful type of neural network designed to handle sequence dependence is called recurrent neural networks. The Long Short-Term Memory network or LSTM network is a type of recurrent neural network used in deep learning because very large architectures can be successfully trained. A Long Short-term memory for time series classification (LSTM). An LSTM is the extension of the classical Recurrent Neural Network. It has more flexibility and interpretable features such as a memory it can read, write and forget.

# Perceptrons

What is a neural network? To get started, I'll explain a type of artificial neuron called a perceptron. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Today, it's more common to use other models of artificial neurons, and in much modern work on neural networks, the main neuron model used is one called the sigmoid neuron. We'll get to sigmoid neurons shortly. But to understand why sigmoid neurons are defined the way they are, it's worth taking the time to first understand perceptrons.

So how do perceptrons work? A perceptron takes several binary inputs, x1,x2,…, and produces a single binary output:

In the example shown the perceptron has three inputs, x1,x2,x3. In general it could have more or fewer inputs. Rosenblatt proposed a simple rule to compute the output. He introduced weights, w1,w2,… real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted sum ∑jwjxj is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms:

(1)

That's all there is to how a perceptron works!

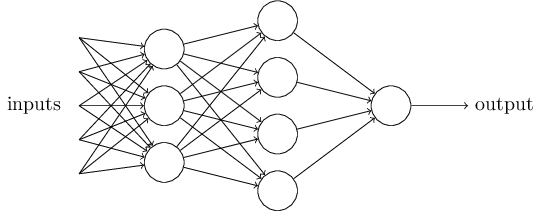
That's the basic mathematical model. A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence. Let me give an example. It's not a very realistic example, but it's easy to understand, and we'll soon get to more realistic examples. Suppose the weekend is coming up, and you've heard that there's going to be a cheese festival in your city. You like cheese, and are trying to decide whether or not to go to the festival. You might make your decision by weighing up three factors:

1. Is the weather good?
2. Does your boyfriend or girlfriend want to accompany you?
3. Is the festival near public transit? (You don't own a car).

We can represent these three factors by corresponding binary variables x1, x2 and x3. For instance, we'd have x1=1 if the weather is good, and x1=0x1=0 if the weather is bad. Similarly, x2=1 if your boyfriend or girlfriend wants to go, and x2=0 if not. And similarly again for x3 and public transit.

Now, suppose you absolutely adore cheese, so much so that you're happy to go to the festival even if your boyfriend or girlfriend is uninterested and the festival is hard to get to. But perhaps you really loathe bad weather, and there's no way you'd go to the festival if the weather is bad. You can use perceptrons to model this kind of decision-making. One way to do this is to choose a weight w1=6 for the weather, and w2=2 and w3=2 for the other conditions. The larger value of w1 indicates that the weather matters a lot to you, much more than whether your boyfriend or girlfriend joins you, or the nearness of public transit. Finally, suppose you choose a threshold of 5 for the perceptron. With these choices, the perceptron implements the desired decision-making model, outputting 1 whenever the weather is good, and 0 whenever the weather is bad. It makes no difference to the output whether your boyfriend or girlfriend wants to go, or whether public transit is nearby.

By varying the weights and the threshold, we can get different models of decision-making. For example, suppose we instead chose a threshold of 3. Then the perceptron would decide that you should go to the festival whenever the weather was good or when both the festival was near public transit and your boyfriend or girlfriend was willing to join you. In other words, it'd be a different model of decision-making. Dropping the threshold means you're more willing to go to the festival.

Obviously, the perceptron isn't a complete model of human decision-making! But what the example illustrates is how a perceptron can weigh up different kinds of evidence in order to make decisions. And it should seem plausible that a complex network of perceptrons could make quite subtle decisions:

In this network, the first column of perceptrons - what we'll call the first layer of perceptrons - is making three very simple decisions, by weighing the input evidence. What about the perceptrons in the second layer? Each of those perceptrons is making a decision by weighing up the results from the first layer of decision-making. In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer. And even more complex decisions can be made by the perceptron in the third layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

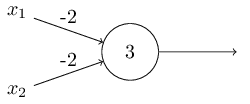
Incidentally, when I defined perceptrons I said that a perceptron has just a single output. In the network above the perceptrons look like they have multiple outputs. In fact, they're still single output. The multiple output arrows are merely a useful way of indicating that the output from a perceptron is being used as the input to several other perceptrons. It's less unwieldy than drawing a single output line which then splits.

Let's simplify the way we describe perceptrons. The condition ∑jwjxj > threshold is cumbersome, and we can make two notational changes to simplify it. The first change is to write ∑jwjxj s a dot product, w⋅x ≡ ∑jwjxj, where w and xx are vectors whose components are the weights and inputs, respectively. The second change is to move the threshold to the other side of the inequality, and to replace it by what's known as the perceptron's bias, b ≡ −threshold. Using the bias instead of the threshold, the perceptron rule can be rewritten:

(2)

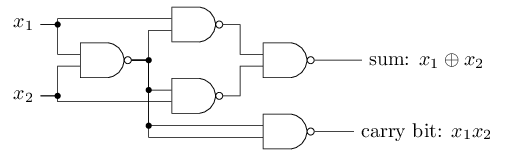
You can think of the bias as a measure of how easy it is to get the perceptron to output a 11. Or to put it in more biological terms, the bias is a measure of how easy it is to get the perceptron to fire. For a perceptron with a really big bias, it's extremely easy for the perceptron to output a 11. But if the bias is very negative, then it's difficult for the perceptron to output a 11. Obviously, introducing the bias is only a small change in how we describe perceptrons, but we'll see later that it leads to further notational simplifications. Because of this, in the remainder of the book we won't use the threshold, we'll always use the bias.

I've described perceptrons as a method for weighing evidence to make decisions. Another way perceptrons can be used is to compute the elementary logical functions we usually think of as underlying computation, functions such as AND, OR, and NAND. For example, suppose we have a perceptron with two inputs, each with weight −2−2, and an overall bias of 33. Here's our perceptron:

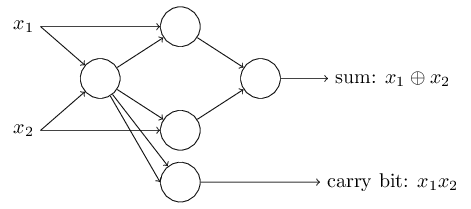


Then we see that input 000 produces output 11, since (−2)∗0+(−2)∗0+3=3 is positive. Here, I've introduced the ∗∗symbol to make the multiplications explicit. Similar calculations show that the inputs 0101 and 1010 produce output 11. But the input 1111produces output 00, since (−2)∗1+(−2)∗1+3=−1 is negative. And so our perceptron implements a NAND gate!

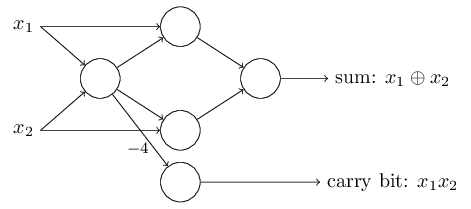
The NAND example shows that we can use perceptrons to compute simple logical functions. In fact, we can use networks of perceptrons to compute *any* logical function at all. The reason is that the NAND gate is universal for computation, that is, we can build any computation up out of NAND gates. For example, we can use NAND gates to build a circuit which adds two bits, x1x1 and x2x2. This requires computing the bitwise sum, x1⊕x2x1⊕x2, as well as a carry bit which is set to 11 when both x1x1 and x2x2 are 11, i.e., the carry bit is just the bitwise product x1x2x1x2:



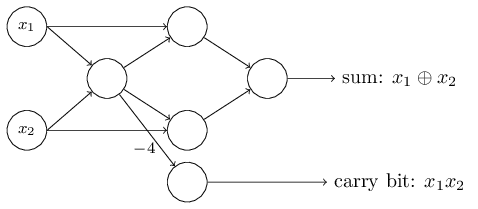
To get an equivalent network of perceptrons we replace all the NANDgates by perceptrons with two inputs, each with weight −2−2, and an overall bias of 33. Here's the resulting network. Note that I've moved the perceptron corresponding to the bottom right NAND gate a little, just to make it easier to draw the arrows on the diagram:



One notable aspect of this network of perceptrons is that the output from the leftmost perceptron is used twice as input to the bottommost perceptron. When I defined the perceptron model I didn't say whether this kind of double-output-to-the-same-place was allowed. Actually, it doesn't much matter. If we don't want to allow this kind of thing, then it's possible to simply merge the two lines, into a single connection with a weight of -4 instead of two connections with -2 weights. (If you don't find this obvious, you should stop and prove to yourself that this is equivalent.) With that change, the network looks as follows, with all unmarked weights equal to -2, all biases equal to 3, and a single weight of -4, as marked:



Up to now I've been drawing inputs like x1x1 and x2x2 as variables floating to the left of the network of perceptrons. In fact, it's conventional to draw an extra layer of perceptrons - the input layer- to encode the inputs:



This notation for input perceptrons, in which we have an output, but no inputs,http://neuralnetworksanddeeplearning.com/images/tikz7.png is a shorthand. It doesn't actually mean a perceptron with no inputs. To see this, suppose we did have a perceptron with no inputs. Then the weighted sum ∑jwjxj would always be zero, and so the perceptron would output 11 if b>0b>0, and 00 if b≤0b≤0. That is, the perceptron would simply output a fixed value, not the desired value (x1x1, in the example above). It's better to think of the input perceptrons as not really being perceptrons at all, but rather special units which are simply defined to output the desired values, x1,x2,…x1,x2,….

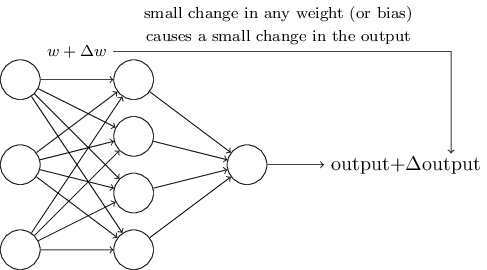
The adder example demonstrates how a network of perceptrons can be used to simulate a circuit containing many NAND gates. And because NAND gates are universal for computation, it follows that perceptrons are also universal for computation.

The computational universality of perceptrons is simultaneously reassuring and disappointing. It's reassuring because it tells us that networks of perceptrons can be as powerful as any other computing device. But it's also disappointing, because it makes it seem as though perceptrons are merely a new type of NAND gate. That's hardly big news!

However, the situation is better than this view suggests. It turns out that we can devise learning algorithms which can automatically tune the weights and biases of a network of artificial neurons. This tuning happens in response to external stimuli, without direct intervention by a programmer. These learning algorithms enable us to use artificial neurons in a way which is radically different to conventional logic gates. Instead of explicitly laying out a circuit of NAND and other gates, our neural networks can simply learn to solve problems, sometimes problems where it would be extremely difficult to directly design a conventional circuit.

# Sigmoid neurons

Learning algorithms sound terrific. But how can we devise such algorithms for a neural network? Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem. For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit. To see how learning might work, suppose we make a small change in some weight (or bias) in the network. What we'd like is for this small change in weight to cause only a small corresponding change in the output from the network. As we'll see in a moment, this property will make learning possible. Schematically, here's what we want (obviously this network is too simple to do handwriting recognition!):

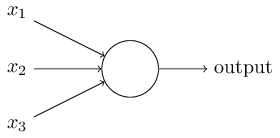


If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want. For example, suppose the network was mistakenly classifying an image as an "8" when it should be a "9". We could figure out how to make a small change in the weights and biases so the network gets a little closer to classifying the image as a "9". And then we'd repeat this, changing the weights and biases over and over to produce better and better output. The network would be learning.

The problem is that this isn't what happens when our network contains perceptrons. In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 00 to 11. That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way. So while your "9" might now be classified correctly, the behaviour of the network on all the other images is likely to have completely changed in some hard-to-control way. That makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to the desired behaviour. Perhaps there's some clever way of getting around this problem. But it's not immediately obvious how we can get a network of perceptrons to learn.

We can overcome this problem by introducing a new type of artificial neuron called a sigmoid neuron. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.

Okay, let me describe the sigmoid neuron. We'll depict sigmoid neurons in the same way we depicted perceptrons:



Just like a perceptron, the sigmoid neuron has inputs, x1,x2,…x1,x2,…. But instead of being just 00 or 11, these inputs can also take on any values *between* 00 and 11. So, for instance, 0.638…0.638… is a valid input for a sigmoid neuron. Also just like a perceptron, the sigmoid neuron has weights for each input, w1,w2,…, and an overall bias, bb. But the output is not 00 or 11. Instead, it's σ(w⋅x+b), where σ is called the sigmoid function\*, and is defined by:

σ(z)≡11+e−z.(3)

\*Incidentally, σσ is sometimes called the *logistic function*, and this new class of neurons called *logistic neurons*. It's useful to remember this terminology, since these terms are used by many people working with neural nets. However, we'll stick with the sigmoid terminology.

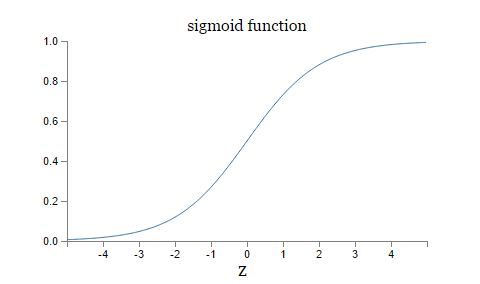
To put it all a little more explicitly, the output of a sigmoid neuron with inputs x1,x2,…x1,x2,…, weights w1,w2,…w1,w2,…, and bias bb is

11+exp(−∑jwjxj−b).(4)

At first sight, sigmoid neurons appear very different to perceptrons. The algebraic form of the sigmoid function may seem opaque and forbidding if you're not already familiar with it. In fact, there are many similarities between perceptrons and sigmoid neurons, and the algebraic form of the sigmoid function turns out to be more of a technical detail than a true barrier to understanding.

To understand the similarity to the perceptron model, suppose z≡w⋅x+bz≡w⋅x+b is a large positive number. Then e−z≈0e−z≈0 and so σ(z)≈1σ(z)≈1. In other words, when z=w⋅x+bz=w⋅x+b is large and positive, the output from the sigmoid neuron is approximately 11, just as it would have been for a perceptron. Suppose on the other hand that z=w⋅x+bz=w⋅x+b is very negative. Then e−z→∞e−z→∞, and σ(z)≈0σ(z)≈0. So when z=w⋅x+bz=w⋅x+b is very negative, the behaviour of a sigmoid neuron also closely approximates a perceptron. It's only when w⋅x+bw⋅x+b is of modest size that there's much deviation from the perceptron model.

What about the algebraic form of σσ? How can we understand that? In fact, the exact form of σσ isn't so important - what really matters is the shape of the function when plotted. Here's the shape:



This shape is a smoothed out version of a step function:

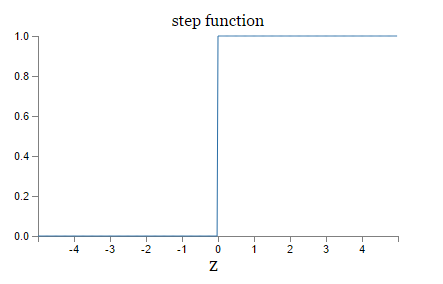
If σσ had in fact been a step function, then the sigmoid neuron would be a perceptron, since the output would be 11 or 00 depending on whether w⋅x+bw⋅x+b was positive or negative\*\*Actually, when w⋅x+b=0w⋅x+b=0 the perceptron outputs 00, while the step function outputs 11. So, strictly speaking, we'd need to modify the step function at that one point. But you get the idea.. By using the actual σσfunction we get, as already implied above, a smoothed out perceptron. Indeed, it's the smoothness of the σσ function that is the crucial fact, not its detailed form. The smoothness of σσ means that small changes ΔwjΔwj in the weights and ΔbΔb in the bias will produce a small change ΔoutputΔoutput in the output from the neuron. In fact, calculus tells us that ΔoutputΔoutput is well approximated by

Δoutput≈∑j∂output∂wjΔwj+∂output∂bΔb,(5)(5)Δoutput≈∑j∂output∂wjΔwj+∂output∂bΔb,

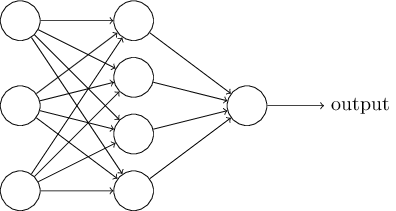
where the sum is over all the weights, wjwj, and ∂output/∂wj∂output/∂wj and ∂output/∂b∂output/∂b denote partial derivatives of the outputoutput with respect to wjwj and bb, respectively. Don't panic if you're not comfortable with partial derivatives! While the expression above looks complicated, with all the partial derivatives, it's actually saying something very simple (and which is very good news): ΔoutputΔoutput is a linear function of the changes ΔwjΔwj and ΔbΔb in the weights and bias. This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output. So while sigmoid neurons have much of the same qualitative behaviour as perceptrons, they make it much easier to figure out how changing the weights and biases will change the output.

If it's the shape of σσ which really matters, and not its exact form, then why use the particular form used for σσ in Equation (3)? In fact, later in the book we will occasionally consider neurons where the output is f(w⋅x+b)f(w⋅x+b) for some other activation function f(⋅)f(⋅). The main thing that changes when we use a different activation function is that the particular values for the partial derivatives in Equation (5) change. It turns out that when we compute those partial derivatives later, using σσ will simplify the algebra, simply because exponentials have lovely properties when differentiated. In any case, σσ is commonly-used in work on neural nets, and is the activation function we'll use most often in this book.

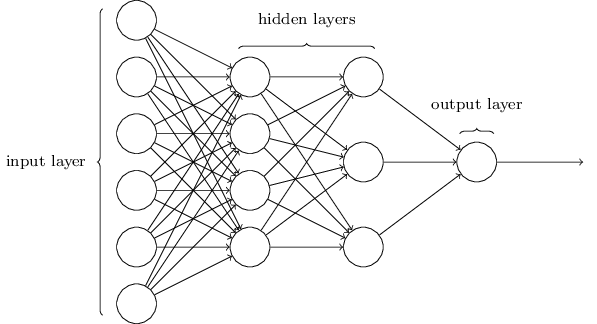
How should we interpret the output from a sigmoid neuron? Obviously, one big difference between perceptrons and sigmoid neurons is that sigmoid neurons don't just output 00 or 11. They can have as output any real number between 00 and 11, so values such as 0.173…0.173… and 0.689…0.689… are legitimate outputs. This can be useful, for example, if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network. But sometimes it can be a nuisance. Suppose we want the output from the network to indicate either "the input image is a 9" or "the input image is not a 9". Obviously, it'd be easiest to do this if the output was a 00 or a 11, as in a perceptron. But in practice we can set up a convention to deal with this, for example, by deciding to interpret any output of at least 0.50.5 as indicating a "9", and any output less than 0.50.5 as indicating "not a 9". I'll always explicitly state when we're using such a convention, so it shouldn't cause any confusion.

The architecture of neural networks

In the next section I'll introduce a neural network that can do a pretty good job classifying handwritten digits. In preparation for that, it helps to explain some terminology that lets us name different parts of a network. Suppose we have the network:



As mentioned earlier, the leftmost layer in this network is called the input layer, and the neurons within the layer are called *input neurons*. The rightmost or *output* layer contains the *output neurons*, or, as in this case, a single output neuron. The middle layer is called a *hidden layer*, since the neurons in this layer are neither inputs nor outputs. The term "hidden" perhaps sounds a little mysterious - the first time I heard the term I thought it must have some deep philosophical or mathematical significance - but it really means nothing more than "not an input or an output". The network above has just a single hidden layer, but some networks have multiple hidden layers. For example, the following four-layer network has two hidden layers:



Somewhat confusingly, and for historical reasons, such multiple layer networks are sometimes called *multilayer perceptrons* or*MLPs*, despite being made up of sigmoid neurons, not perceptrons. I'm not going to use the MLP terminology in this book, since I think it's confusing, but wanted to warn you of its existence.

The design of the input and output layers in a network is often straightforward. For example, suppose we're trying to determine whether a handwritten image depicts a "9" or not. A natural way to design the network is to encode the intensities of the image pixels into the input neurons. If the image is a 64 by 64 greyscale image, then we'd have 4,096=64×64 input neurons, with the intensities scaled appropriately between 0 and 1. The output layer will contain just a single neuron, with output values of less than 0.5 indicating "input image is not a 9", and values greater than 0.5 indicating "input image is a 9 ".

While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers. In particular, it's not possible to sum up the design process for the hidden layers with a few simple rules of thumb. Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behaviour they want out of their nets. For example, such heuristics can be used to help determine how to trade off the number of hidden layers against the time required to train the network. We'll meet several such design heuristics later in this book.

Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer. Such networks are called *feedforward* neural networks. This means there are no loops in the network - information is always fed forward, never fed back. If we did have loops, we'd end up with situations where the input to the σσ function depended on the output. That'd be hard to make sense of, and so we don't allow such loops.

However, there are other models of artificial neural networks in which feedback loops are possible. These models are called[recurrent neural networks](http://en.wikipedia.org/wiki/Recurrent_neural_network). The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent. That firing can stimulate other neurons, which may fire a little while later, also for a limited duration. That causes still more neurons to fire, and so over time we get a cascade of neurons firing. Loops don't cause problems in such a model, since a neuron's output only affects its input at some later time, not instantaneously.

Recurrent neural nets have been less influential than feedforward networks, in part because the learning algorithms for recurrent nets are (at least to date) less powerful. But recurrent networks are still extremely interesting. They're much closer in spirit to how our brains work than feedforward networks. And it's possible that recurrent networks can solve important problems which can only be solved with great difficulty by feedforward networks. However, to limit our scope, in this book we're going to concentrate on the more widely-used feedforward networks.

# Understanding LSTM Networks

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.

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.

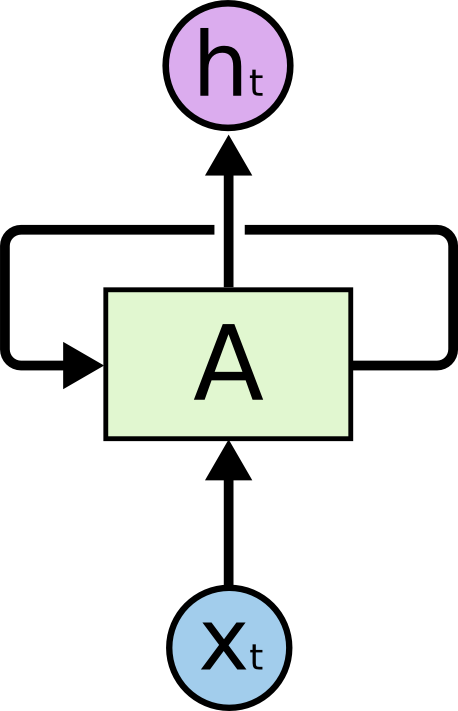


Figure : Recurrent Neural Networks have loops.

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:

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.

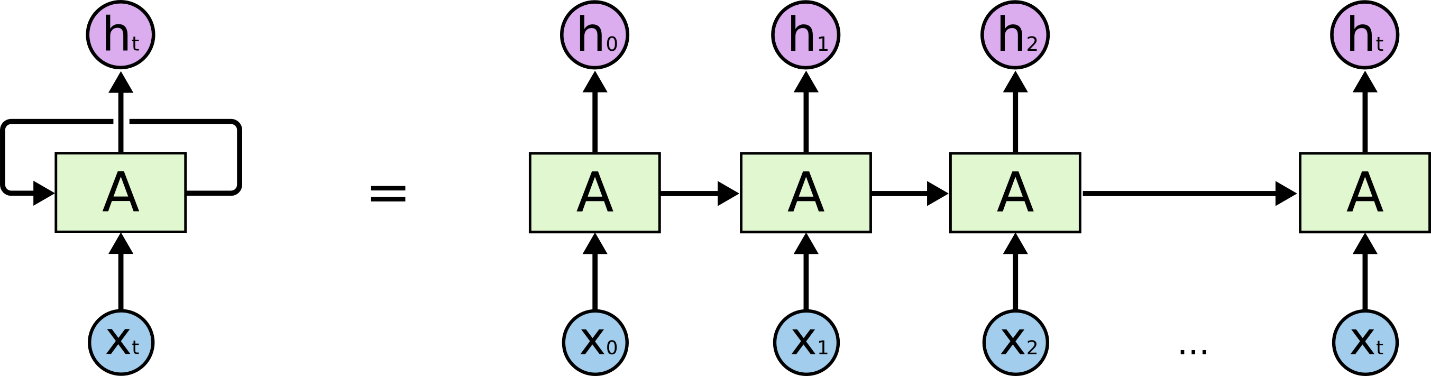


Figure : An unrolled recurrent neural network.

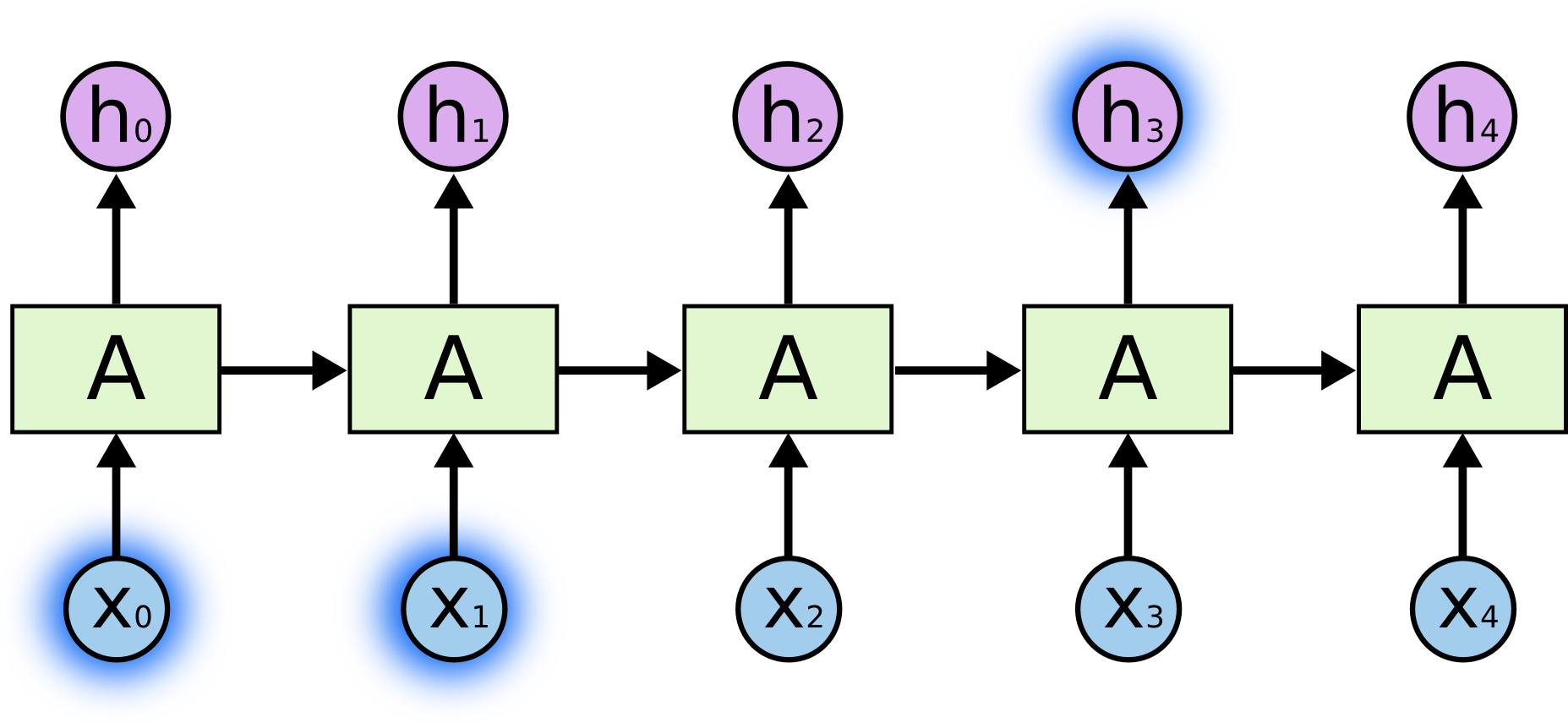
And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning… But they really are pretty amazing.

Essential to these successes is the use of “LSTMs,” a very special kind of recurrent neural network which works, for many tasks, much 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

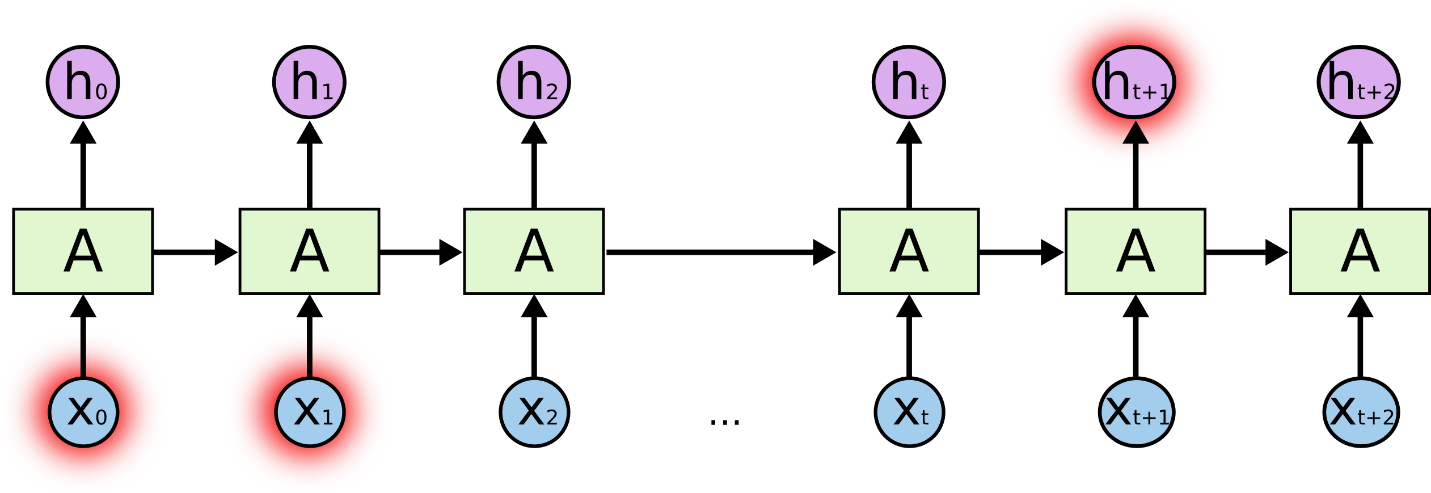
One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they’d be extremely useful. But can they? It depends.

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] and Bengio, et al. (1994), 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), and were refined and popularized by many people in following work. 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 than layer.

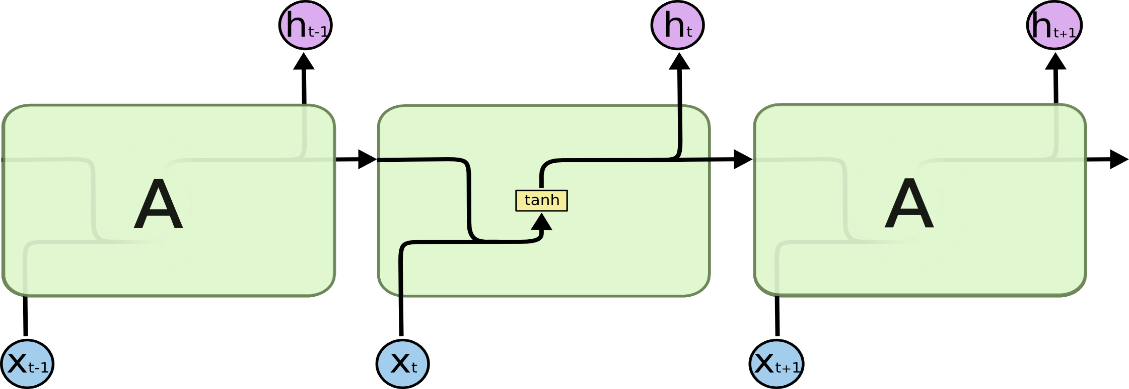


Figure : The repeating module in a standard RNN contains a single layer.

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.

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.

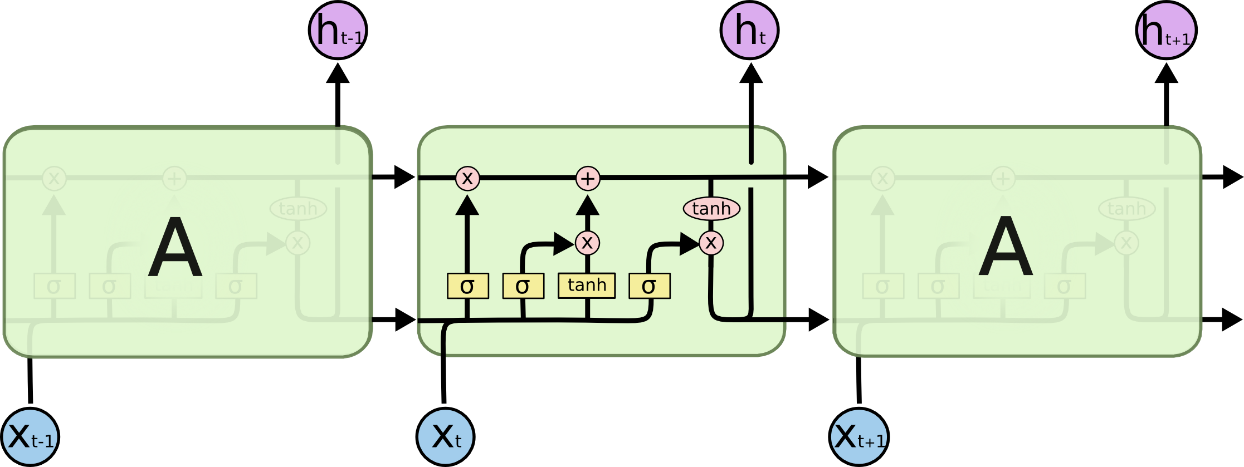
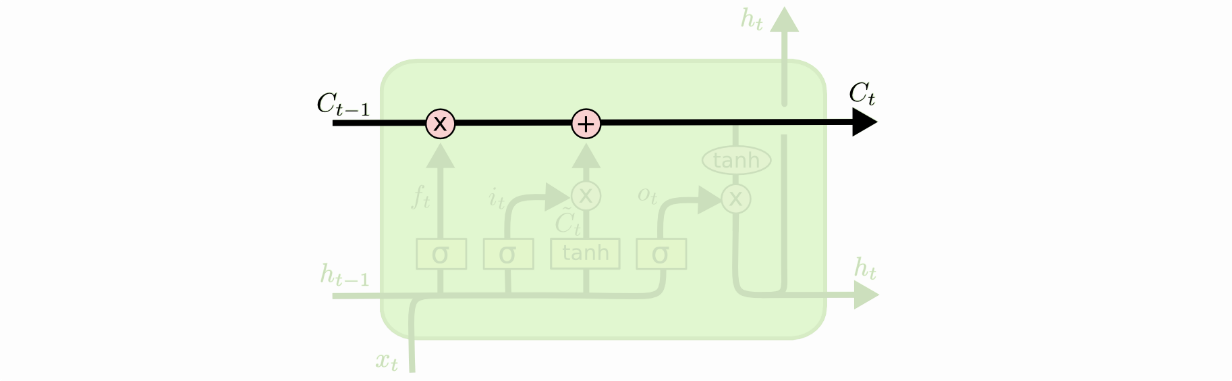


Figure : The repeating module in an LSTM contains four interacting layers.

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−1ht−1and xtxt, 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.



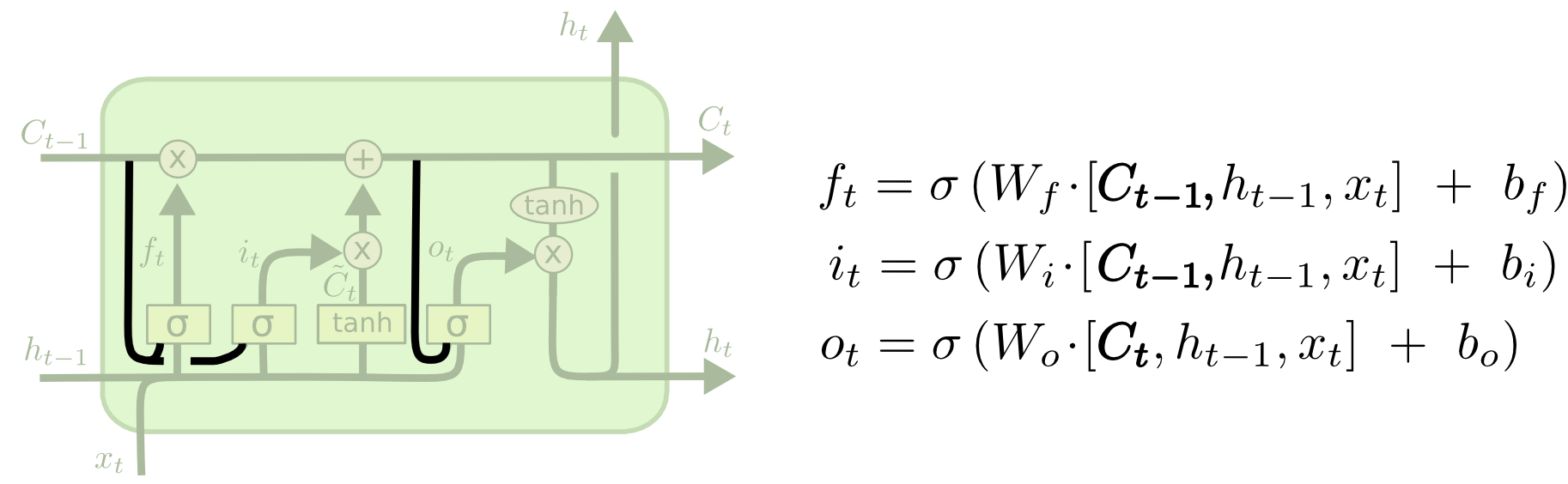
Finally, we need to decide what we’re going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we’re going to output. Then, we put the cell state through tanh (to push the values to be between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

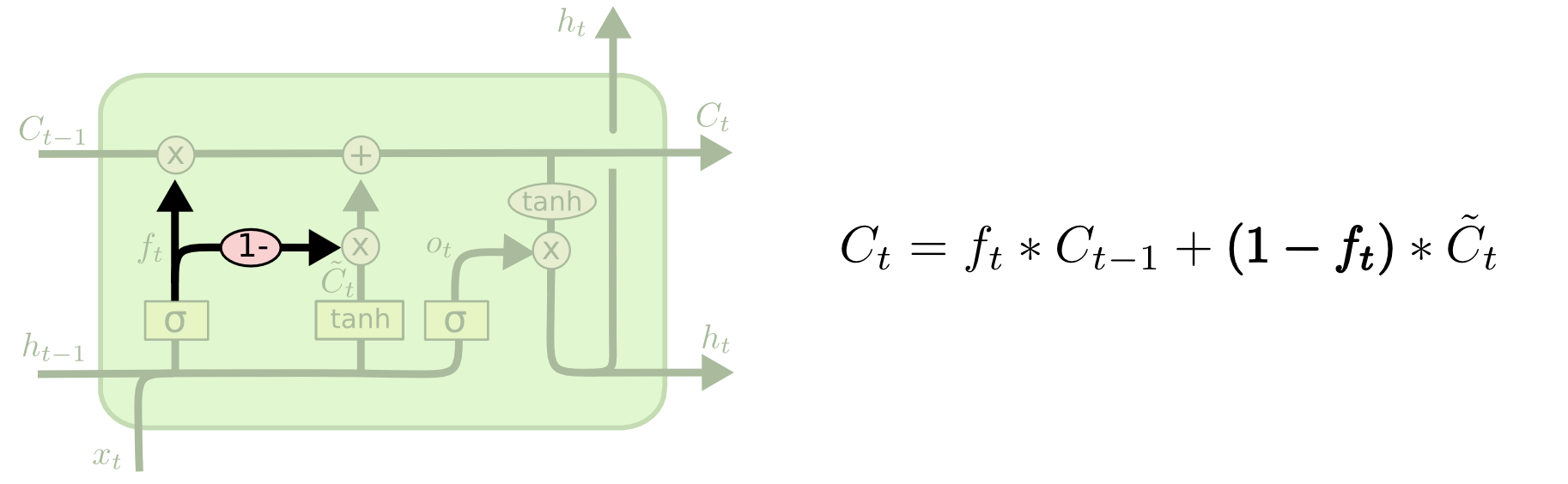
What I’ve described so far is a pretty normal LSTM. But not all LSTMs are the same as the above. In fact, it seems like almost every paper involving LSTMs uses a slightly different version. The differences are minor, but it’s worth mentioning some of them.

One popular LSTM variant, introduced by Gers & Schmidhuber (2000), is adding “peephole connections.” This means that we let the gate layers look at the cell state.

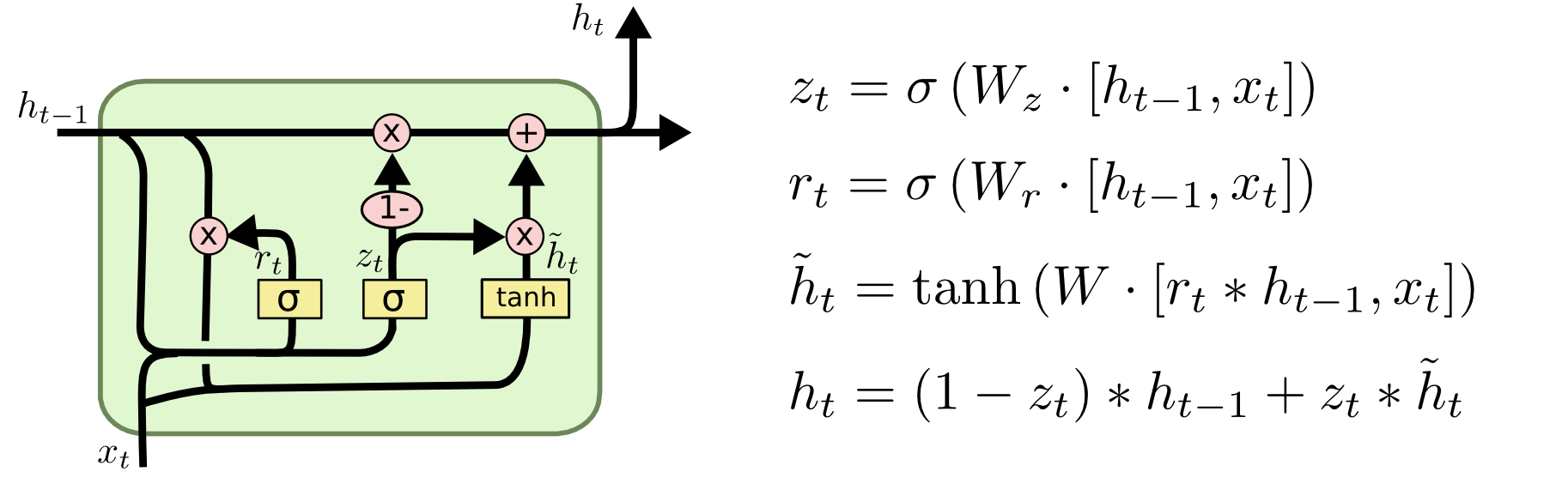


The above diagram adds peepholes to all the gates, but many papers will give some peepholes and not others.

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). It combines 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). There’s also some completely different approach to tackling long-term dependencies, like Clockwork RNNs by Koutnik, et al. (2014).

# Predictions

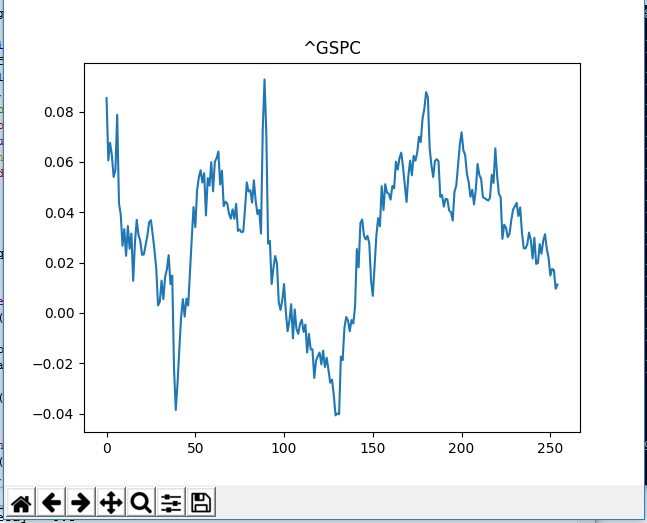
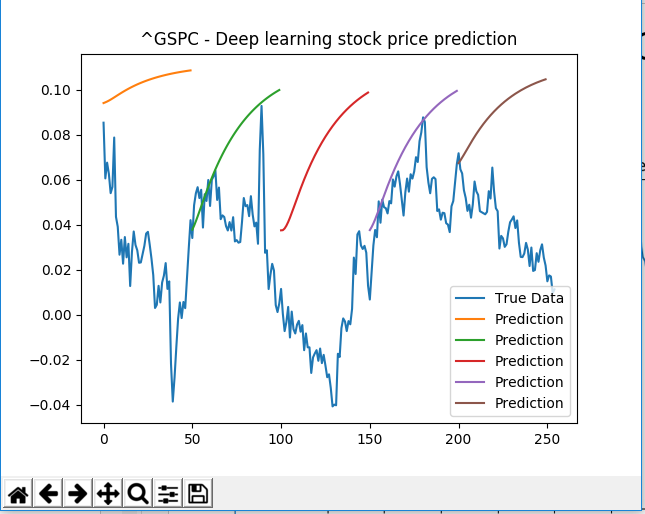


Figure : S&P 500 for 1-Jan-2007 to 8-May-2017



# References

Crawford, C. (2016, November 4). An introduction to deep learning. Retrieved February 5, 2017, from A.I. Topic Guides, <http://blog.algorithmia.com/introduction-to-deep-learning-2016/>

(Crawford, 2016)

Deep Learning. (n.d.). Retrieved March 14, 2017, from <http://www.deeplearningbook.org/>

Santos, L. (2017). Introduction · Artificial Intelligence. Leonardoaraujosantos.gitbooks.io. Retrieved 14 March 2017, from <https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/chapter1.html>

(Santos, 2017)

Nielsen, M. A. (1970, January 01). Neural Networks and Deep Learning. Retrieved May 06, 2017, from <http://neuralnetworksanddeeplearning.com/chap1.html#the_architecture_of_neural_networks>

http://neuralnetworksanddeeplearning.com/chap1.html#the\_architecture\_of\_neural\_networks

<https://arxiv.org/pdf/1607.03085.pdf>

<https://stocksneural.net/about>

<https://www.youtube.com/watch?v=BrPls5sVYfM>

<https://www.youtube.com/watch?v=9aHJ-FAzQaE>

<https://www.youtube.com/watch?v=-zT1Zi_ukSk>

<https://channel9.msdn.com/Events/Build/2014/3-643>

<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

<http://machinelearningmastery.com/inspirational-applications-deep-learning/>

<https://www.technologyreview.com/s/513696/deep-learning/>

<https://github.com/ChristosChristofidis/awesome-deep-learning#videos-and-lectures>

<http://www.datasciencecentral.com/profiles/blogs/evolution-of-deep-learning-models>

<http://www.datasciencecentral.com/profiles/blogs/a-tour-of-machine-learning-algorithms>

<http://www.datasciencecentral.com/profiles/blogs/classifications-in-r-response-modeling-credit-scoring-credit>

<http://www.datasciencecentral.com/profiles/blogs/classifications-in-r-response-modeling-credit-scoring-credit>

<http://www.datasciencecentral.com/page/search?q=deep+learning&cx=007889287169707156432%3A04zouenp6u0&ie=UTF-8&cof=FORID%3A11%3BNB%3A1&sa=Search>