July 2016 Issue # 2

THE MORNING PAPER
QUARTERLY REVIEW

THE AMAZING POWER OF WORD VECTORS

FACEBOOK'S GORILLA: A FAST, SCALABLE, IN-MEMORY TIME-SERIES DATABASE

DEEP LEARNING IN NEURAL NETWORKS: AN OVERVIEW

A selection of papers from the world of computer science, as featured by Adrian Colyer on The Morning Paper blog

InfoQ

CONTENTS

Issue # 2, July 2016

WELCOME...

Adrian Colyer

...TO THE 2ND EDITION OF THE MORNING PAPER QUARTERLY

Through April, May, and June I covered 55 research papers on The Morning Paper, and whittling these down to just five choices was once again a difficult task. With apologies to all of the authors whose wonderful works I had to leave out, here are my five picks for the second quarter, in the order that they originally appeared on the blog.

THE AMAZING POWER OF WORD VECTORS

This choice is really a little bit of a cheat, because the original The Morning Paper blog post on this topic actually drew material from five papers. Chief amongst them is "Efficient estimation of word representations in vector space," by Mikolov et al. 2013. The big idea is that we can learn vectors (think arrays of real numbers) that somewhat amazingly actually come to embody the _meaning_ of a word. In fact, they do this so well that we can even perform vector arithmetic with the representations and get meaningful results. For example, (the vector for) `apples` - (the vector for) `apple` is approximately the same as `cars - car`, and `families - family`, and so on. And `woman - man` is approximately equal to `aunt - uncle`, and `queen - king`. This leads to the often-quoted example that we can ask "what is king - man + woman?" and get back the answer "queen!" The word vector representations learned by the word2vec algorithm have proven to be a very powerful feature representation for all sorts of natural language tasks that build on top of them.

DEEP LEARNING IN NEURAL NETWORKS: AN OVERVIEW

This 2014 paper by Jürgen Schmidhuber packs a huge amount of information into a small space, and has comprehensive references if you want to go deeper in any area. Schmidhuber puts half a century of machine learning into context, tracing the roots of modern deep learning techniques. I found it tremendously helpful in building my own mental map of the space, and in understanding some of the key themes and challenges that transcend any one particular technique.

GORILLA: A FAST, SCALABLE, IN-MEMORY TIME SERIES DATABASE

This is the story of how Facebook collect and analyse over 1 trillion data points per day across 2 billion unique time series. The system they built to do this is called Gorilla, an in-memory time series database. To cope with the volume and velocity, Gorilla uses a novel time series compression algorithm which reduces memory / storage requirements by 12x on average. For those of you who like to get down to bits and bytes, I go into some detail on this in my write-up. Gorilla can re-

spond to queries 70x faster than the system it replaced, and this has enabled the team at Facebook to build a number of tools on top. Of particular interest is the correlation engine that can help with root cause analysis when diagnosing production issues.

HOW TO BUILD STATIC CHECKING SYSTEMS USING ORDERS OF MAGNITUDE LESS CODE

On the surface this is a somewhat unusual choice, as I doubt many of you will have a need to build your own static code checker. But I chose this paper for the general lesson it teaches about the layers of complexity that can build up in a system over time. The authors are to be praised for recognising that in their own work and tools, complexity had steadily grown to the point where they couldn't really understand what was going on anymore, and they certainly couldn't continue to make rapid progress. From the realisation that "there must be a better way," and by daring to ask the question, they were able to come up with a dramatically simpler alternative that performed surprisingly well. Inspiration for us all.

A SURVEY OF AVAILABLE CORPORA FOR BUILDING DATA DRIVEN DIALOG SYSTEMS

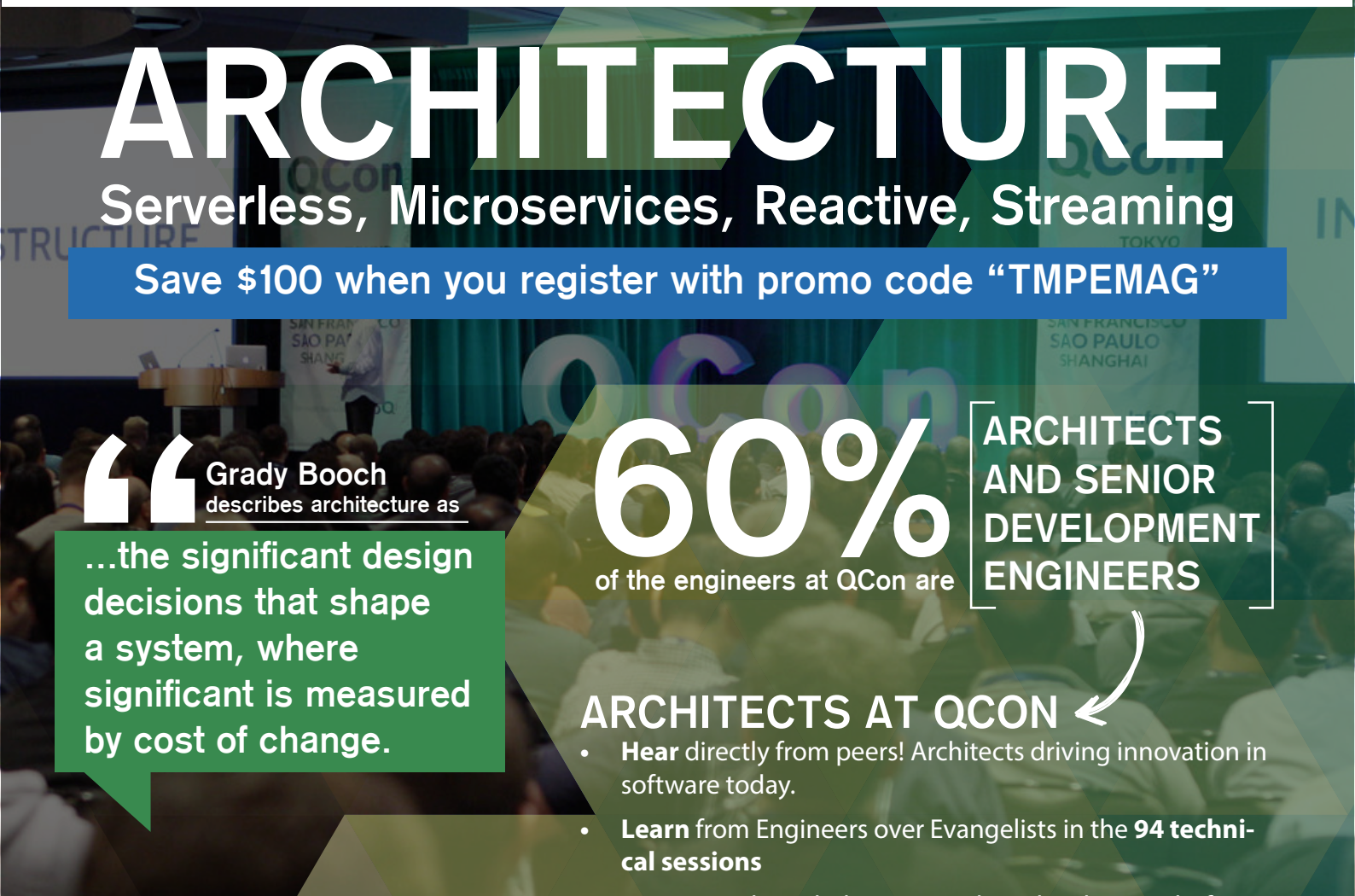
It's not the snappiest title I'll grant you. But 'data driven dialog systems,' aka conversational interfaces, are hot right now. We spend increasing portions of our time inside messaging apps, and chatbots allow you to bring your application to where the users already are (inside Facebook Messenger, or Slack for example). And beyond primarily textual interfaces, dialog systems also power voice-based interactions - think Siri, Cortana, Alexa et al. As of the end of June, over 11,000 bots had been created on Messenger since the launch of the chatbot platform in April. I suspect many of these don't really have any conversational skills to speak of. But if you want to understand what it takes to start building a truly conversational interface, you could do a lot worse than start with "A survey of available corpora for building data driven dialog systems." Along with information about available data sets you can use, this paper also contains a nice overview of the challenges involved in building a dialogue system and pointers to the techniques researchers are using to overcome them.

QCon SAN FRANCISCO
Nov 7-11, 2016

Conference for Professional Software Developers
qconsf.com

ARCHITECTURE
Serverless, Microservices, Reactive, Streaming

Save $100 when you register with promo code "TMPEMAG"



"Grady Booch describes architecture as ...the significant design decisions that shape a system, where significant is measured by cost of change.

60% of the engineers at QCon are ARCHITECTS AND SENIOR DEVELOPMENT ENGINEERS

ARCHITECTS AT QCON

- **Hear** directly from peers! Architects driving innovation in software today.
- **Learn** from Engineers over Evangelists in the **94 technical sessions**
- **Jumpstart** knowledge on trends and tech in tracks focused on Lambda, Microservices, Reactive, & Streaming.
- **Connect and Learn** with other technology innovators and early adopters architects in open spaces, workshops, and hallways tracks.

Director of Operations @Netflix (Josh Evans), is presenting "Mastering Chaos - A Netflix Guide to Microservices" at Qcon SF.

Architecture tracks at QConSF feature engineers exploring those costs and tradeoffs like:

- Slack's Chief Architect in *How Slack is Built*
- Leaders like the co-creator of Kafka Neha Narkhede in *The Rise of Realtime*
- Josh Evans the Director of Operations Engineering in *Mastering Chaos - A Netflix Guide to Microservices*
- And **many more** from companies like: Instagram, Uber, Google, and Twitch

The heart of QCon is architecture and features 4 tracks (over 24 sessions) dedicated to tradeoffs, implementations, and architectural design.

THE AMAZING POWER OF WORD VECTORS

This chapter draws on material not from just one paper but from five! The subject matter is word2vec: the work of Mikolov et al. on efficient vector representations of words (and what you can do with them).

"Efficient Estimation of Word Representations in Vector Space" *Mikolov et al. 2013a*
"Distributed Representations of Words and Phrases and their Compositionality" *Mikolov et al. 2013b*
"Linguistic Regularities in Continuous Space Word Representations" *Mikolov et al. 2013c*
"word2vec Parameter Learning Explained" *Rong 2014*
word2vec Explained: Deriving Mikolov et al.'s Negative-Sampling Word-Embedding Method" *Goldberg and Levy 2014*

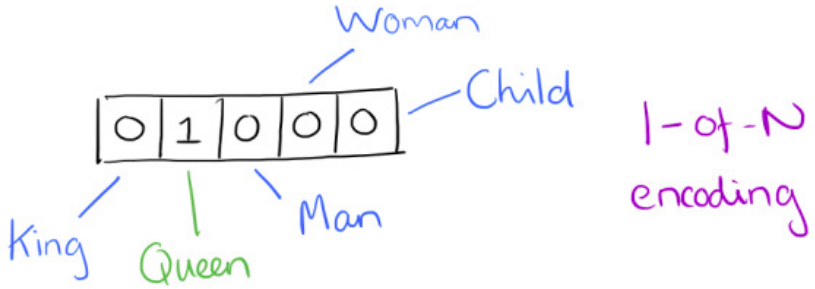
From the first paper on this list, we get a description of the "*continuous bag of words*" and "*continuous skip-gram*" models for learning word vectors (we'll talk about what a word vector is in a moment...). From the second paper, we get more illustrations of the power of word vectors, some additional information on optimisations for the skip-gram model (hierarchical softmax and negative sampling), and a discussion of applying word vectors to phrases. The third paper describes vector-oriented reasoning based on word vectors and introduces the famous "King – Man + Woman = Queen" example. The last two papers give a more detailed explanation of some of the concisely expressed ideas in the Mikolov et al. papers.

Check out the word2vec implementation on Google Code.

What is a word vector?

At one level, a word vector is simply a vector of weights. In a simple 1-of-N (or "one-hot") encoding, every element in the vector is associated with a word in the vocabulary. The encoding of a given word is simply the vector in which the corresponding element is set to one, and all other elements are zero.

Suppose our vocabulary has only five words: King, Queen, Man, Woman, and Child. We could encode the word Queen as:



Using such an encoding, there's no meaningful comparison we can make between word vectors other than equality testing.

But word2vec uses a *distributed* representation of a word. Take a vector with many dimensions — say 1,000. Each word is represented by a distribution of weights across those 1,000 elements. So instead of a one-to-one mapping between an element in the vector and a word, the representation of a word is spread across all of the elements in the vector, and each element in the vector contributes to the definition of many words.

If we label the dimensions in a hypothetical word vector (there are no such pre-assigned labels in the algorithm, of course), it might look a bit like this:

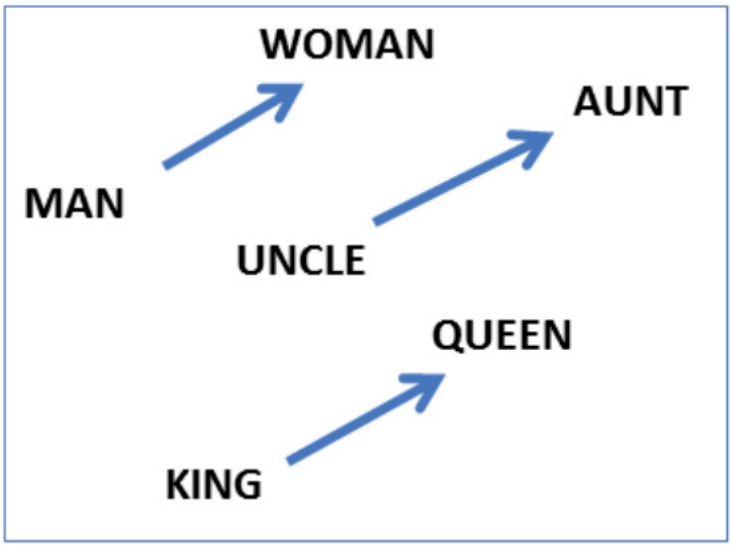
Such a vector comes to represent in some abstract way the meaning of a word. And, as we'll see next, simply by examining a large corpus, it's possible to learn word vectors that are able to capture the relationships between words in a surprisingly expressive way. We can also use the vectors as inputs to a neural network.

Reasoning with word vectors

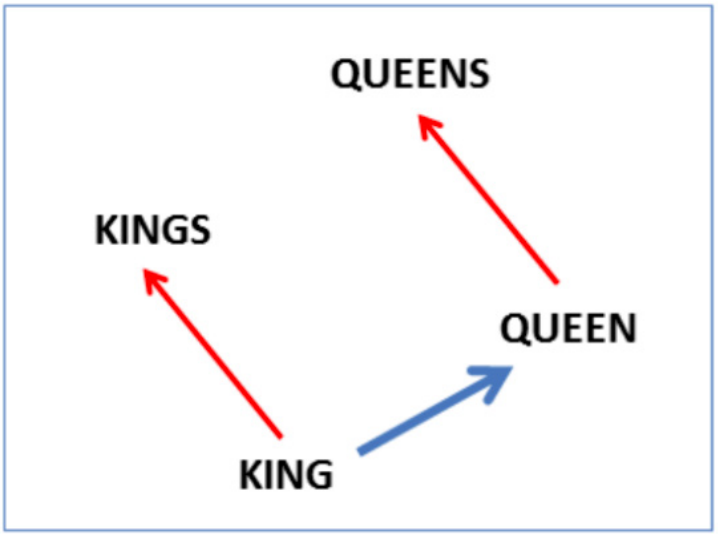
We find that the learned word representations in fact capture meaningful syntactic and semantic regularities in a very simple way. Specifically, the regularities are observed as constant vector offsets between pairs of words sharing a particular relationship. For example, if we denote the vector for word i as xi, and focus on the singular/plural relation, we observe that $x_{apple} - x_{apples} \approx x_{car} - x_{cars}$, $x_{family} - x_{families} \approx x_{car} - x_{cars}$, *and so on. Perhaps more surprisingly, we find that this is also the case for a variety of semantic relations, as measured by the SemEval 2012 task of measuring relation similarity.* ***(Mikolov et al. 2013c)***

The vectors are very good at answering analogy questions of the form "*a* is to *b* as *c* is to *?*" — say, *man* is to *woman* as *uncle* is to *?* (*aunt*) — using a simple vector-offset method based on cosine distance.

For example, here are vector offsets for three word pairs to illustrate the sex relation:



And here we see the singular/plural relation:

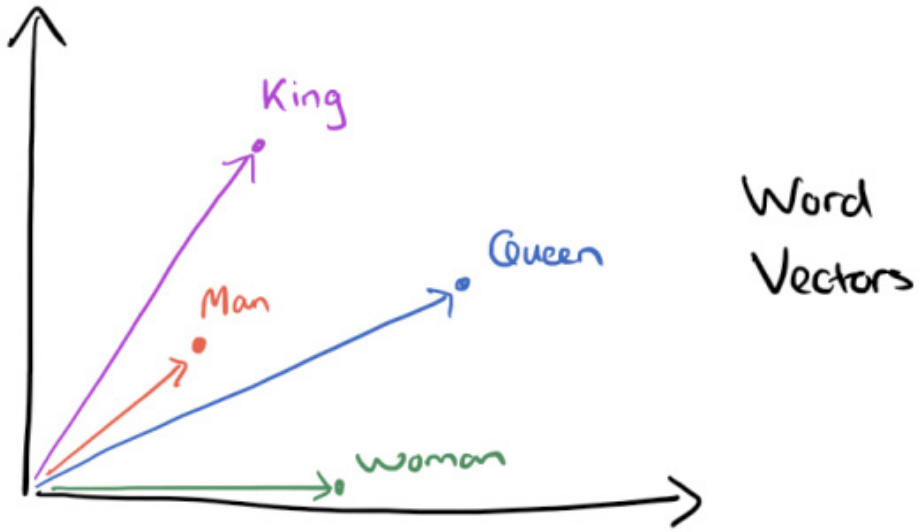


This kind of vector composition also lets us answer "King – Man + Woman = ?" question and arrive at the result "Queen"! All of this is truly remarkable when you think that all of this knowledge simply comes from looking at lots of words in context (as we'll see soon) with no other information provided about their semantics.

> **We find that the learned word representations in fact capture meaningful syntactic and semantic regularities in a very simple way. Specifically, the regularities are observed as constant vector offsets between pairs of words sharing a particular relationship.**

Somewhat surprisingly, it was found that similarity of word representations goes beyond simple syntactic regularities. Using a word-offset technique where simple algebraic operations are performed on the word vectors, it was shown for example that vector("King") – vector("Man") + vector("Woman") results in a vector that is closest to the vector representation of the word "Queen". ***(Mikolov et al. 2013a)***

Word vectors for King, Man, Queen, and Woman are as follows.



And here is the result of the vector composition "King - Man + Woman = ?"

Here are more results achieved using the same technique.

Table 8: *Examples of the word pair relationships, using the best word vectors from Table 4 (Skip-gram model trained on 783M words with 300 dimensionality).*

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

(Mikolov et al. 2013a)

Here's what the country-capital relationship looks like in a two-dimensional PCA projection:

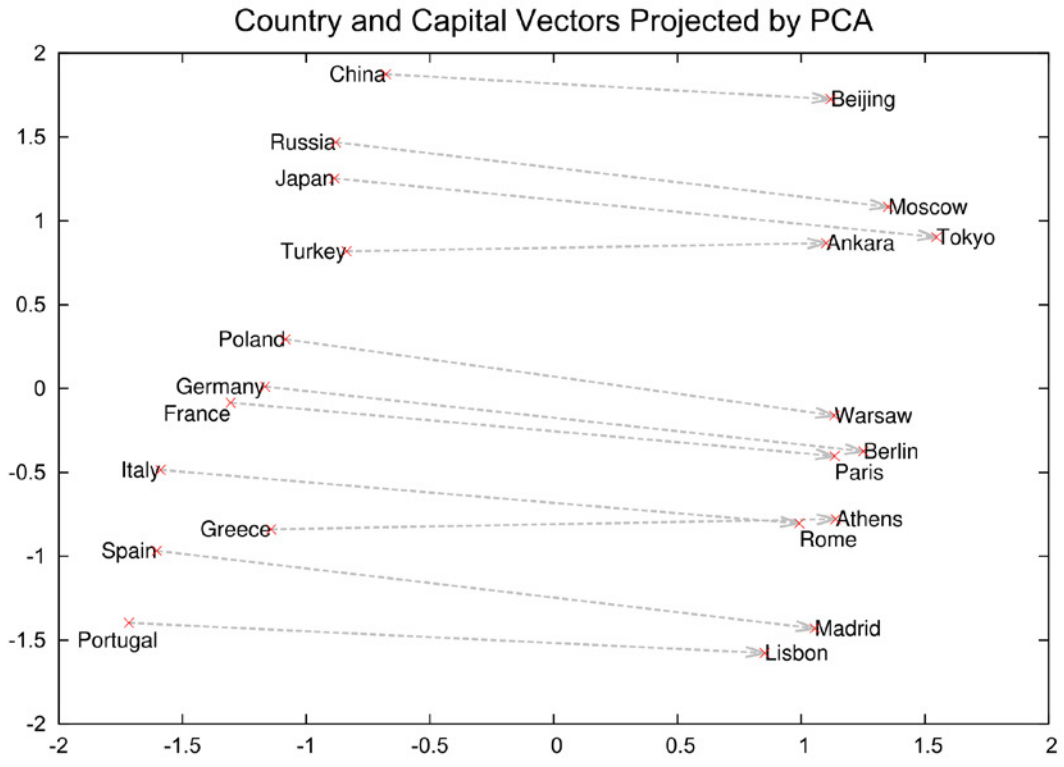


Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised information about what a capital city means.

(Mikolov et al. 2013b)

Here are more examples of the "a is to b as c is to ?" style of question answered by word vectors.

Newspapers			
New York	New York Times	Baltimore	Baltimore Sun
San Jose	San Jose Mercury News	Cincinnati	Cincinnati Enquirer
NHL Teams			
Boston	Boston Bruins	Montreal	Montreal Canadiens
Phoenix	Phoenix Coyotes	Nashville	Nashville Predators
NBA Teams			
Detroit	Detroit Pistons	Toronto	Toronto Raptors
Oakland	Golden State Warriors	Memphis	Memphis Grizzlies
Airlines			
Austria	Austrian Airlines	Spain	Spainair
Belgium	Brussels Airlines	Greece	Aegean Airlines
Company executives			
Steve Ballmer	Microsoft	Larry Page	Google
Samuel J. Palmisano	IBM	Werner Vogels	Amazon

Table 2: Examples of the analogical reasoning task for phrases (the full test set has 3218 examples). The goal is to compute the fourth phrase using the first three. Our best model achieved an accuracy of 72% on this dataset.

(Mikolov et al. 2013b)

We can also use element-wise addition of vector elements to ask questions such as "German + airlines = ?" and by looking at the closest tokens to the composite vector come up with impressive answers:

Czech + currency	Vietnam + capital	German + airlines	Russian + river	French + actress
koruna	Hanoi	airline Lufthansa	Moscow	Juliette Binoche
Check crown	Ho Chi Minh City	carrier Lufthansa	Volga River	Vanessa Paradis
Polish zolty	Viet Nam	flag carrier Lufthansa	upriver	Charlotte Gainsbourg
CTK	Vietnamese	Lufthansa	Russia	Cecile De

Table 5: Vector compositionality using element-wise addition. Four closest tokens to the sum of two vectors are shown, using the best Skip-gram model.

(Mikolov et al. 2013b)

Word vectors with such semantic relationships could be used to improve many existing NLP applications, such as machine translation, information retrieval and question answering systems, and may enable other future applications yet to be invented. ***(Mikolov et al. 2013a)***

The semantic-syntactic word-relationship tests for understanding of a wide variety of relationships is shown below. Using 640-dimensional word vectors, a skip-gram-trained model achieved 55% semantic accuracy and 59% syntactic accuracy.

Table 3: *Comparison of architectures using models trained on the same data, with 640-dimensional word vectors. The accuracies are reported on our Semantic-Syntactic Word Relationship test set, and on the syntactic relationship test set of [20]*

Model Architecture	Semantic-Syntactic Word Relationship test set		MSR Word Relatedness Test Set [20]
	Semantic Accuracy [%]	Syntactic Accuracy [%]	
RNNLM	9	36	35
NNLM	23	53	47
CBOW	24	64	61
Skip-gram	55	59	56

(Mikolov et al. 2013a)

Learning word vectors

Mikolov et al. weren't the first to use continuous vector representations of words, but they did show how to reduce the computational complexity of learning such representations, making it practical to learn high-dimensional word vectors on a large amount of data. For example, they stated in the first paper (Mikolov et al. 2013a) that "We have used a Google News corpus for training the word vectors. This corpus contains about 6B tokens. We have restricted the vocabulary size to the 1 million most frequent words."

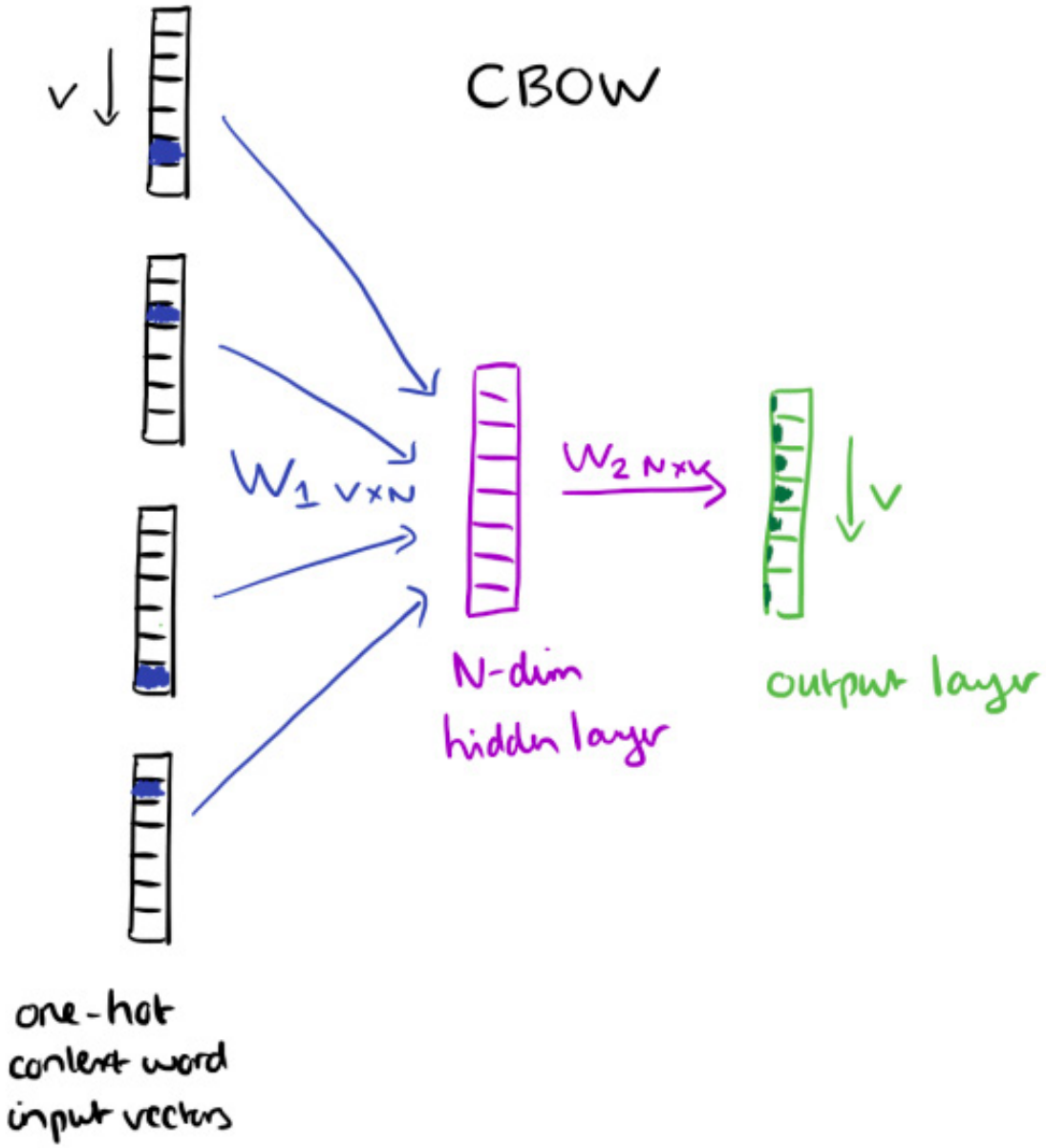
The complexity in neural-network language models (feed forward or recurrent) comes from the non-linear hidden layer(s).

While this is what makes neural networks so attractive, we decided to explore simpler models that might not be able to represent the data as precisely as neural networks, but can possibly be trained on much more data efficiently. ***(Mikolov et al. 2013a)***

Mikolov et al. propose two new architectures: a *continuous bag-of-words* (CBOW) model and a *continuous skip-gram* model. Let's look at the CBOW model first.

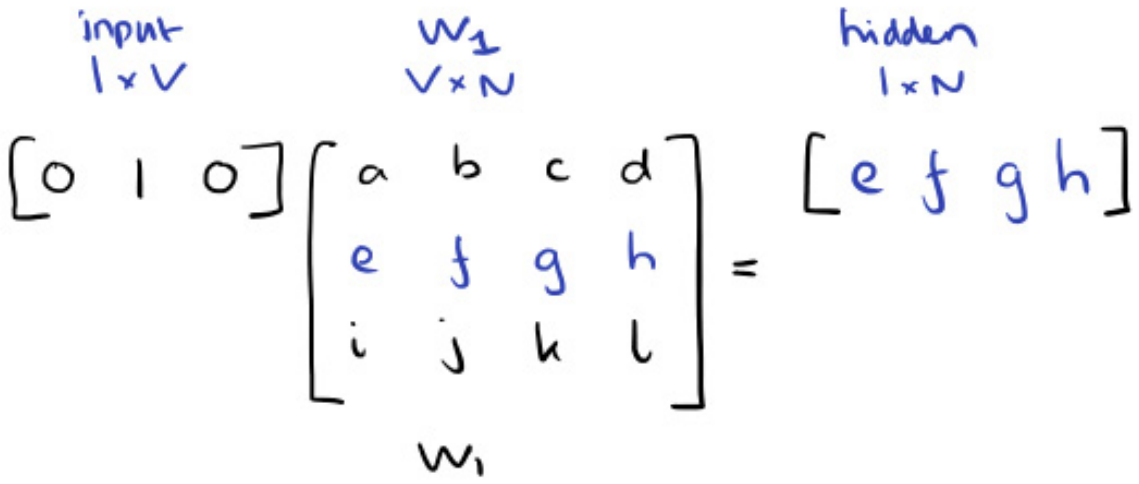
Consider a piece of prose such as "The recently introduced continuous Skip-gram model is an efficient method for learning high-quality distributed vector representations that capture a large number of precise syntactic and semantic word relationships." Imagine a window sliding over the text to include a central word currently in focus together with the four words and precede it and the four words that follow it:

The context words form the input layer. Each word is encoded in one-hot form, so if the vocabulary size is *V* these will be V-dimensional vectors with just one of the elements set to one, and the rest all zeros. There is a single hidden layer and an output layer.



The training objective is to maximize the conditional probability of observing the actual output word (the focus word) given the input context words, with regard to the weights. In our example, given the input ("an", "efficient", "method", "for", "high", "quality", "distributed", "vector"), we want to maximize the probability of getting "learning" as the output.

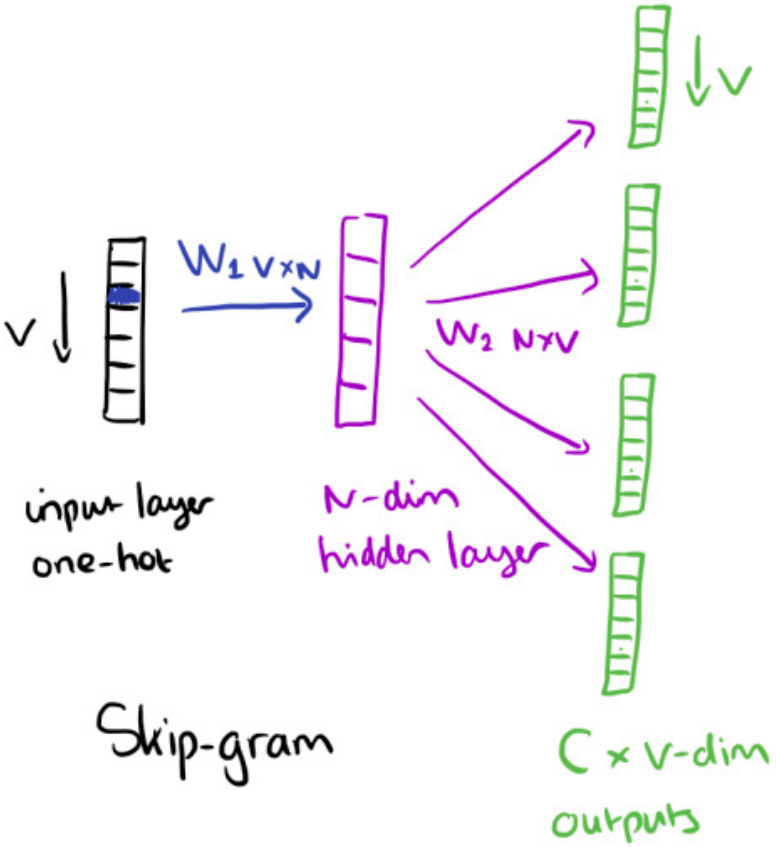
Since our input vectors are one-hot, multiplying an input vector by the weight matrix **W1** amounts to simply selecting a row from **W1**.



Given *C* input word vectors, the activation function for the hidden layer amounts to simply summing the corresponding "hot" rows in W1 and dividing by C to take their mean average.

This implies that the link (activation) function of the hidden layer units is simply linear (i.e., directly passing its weighted sum of inputs to the next layer). ***(Rong 2014)***

From the hidden layer to the output layer, the second weight matrix **W2**can be used to compute a score for each word in the vocabulary, and softmax can be used to obtain the posterior distribution of words.



The **skip-gram** model is the opposite of the CBOW model. It is constructed with the focus word as the single input vector, and the target context words are now at the output layer:

The activation function for the hidden layer simply amounts to copying the corresponding row from the weights matrix **W1** (linear) as we saw before. At the output layer, we now output *C* multinomial distributions instead of just one. The training objective is to minimize the summed prediction error across all context words in the output layer. In our example, the input would be "learning", and we hope to see ("an", "efficient", "method", "for", "high", "quality", "distributed", "vector") at the output layer.

Optimisations

Having to update every output word vector for every word in a training instance is very expensive.

To solve this problem, an intuition is to limit the number of output vectors that must be updated per training instance. One elegant approach to achieving this is hierarchical softmax; another approach is through sampling.... ***(Rong 2014)***

Hierarchical softmax uses a binary tree to represent all words in the vocabulary. The words themselves are leaves in the tree. For each leaf, there exists a unique path from the root to the leaf, and this path is used to estimate the probability of the word represented by the leaf. "We define this probability as the probability of a random walk starting from the root ending at the leaf unit in question." (Rong 2014)

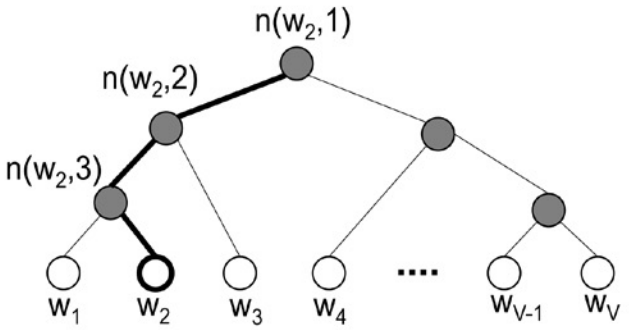


Figure 4: An example binary tree for the hierarchical softmax model. The white units are words in the vocabulary, and the dark units are inner units. An example path from root to w_2 is highlighted. In the example shown, the length of the path $L(w_2) = 4$. $n(w, j)$ means the j-th unit on the path from root to the word w.

(Rong 2014)

The main advantage is that instead of evaluating V output nodes in the neural network to obtain the probability distribution, it is needed to evaluate only about log2(V) words.... In our work we use a binary Huffman tree, as it assigns short codes to the frequent words which results in fast training. ***(Mikolov et al. 2013b)***

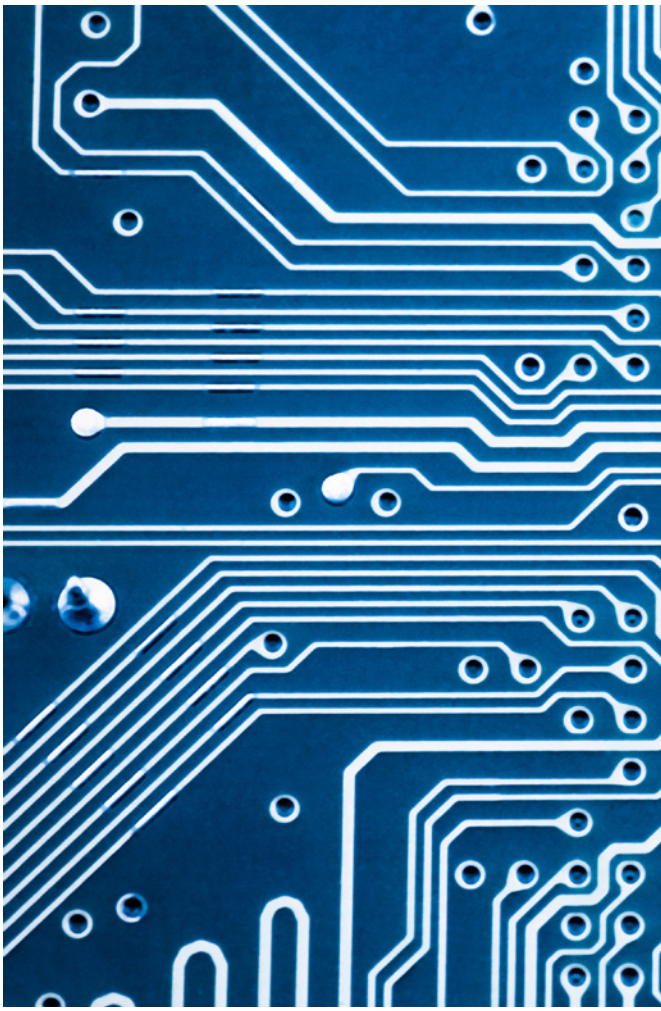
Negative sampling is simply the idea that we only update a sample of output words per iteration. The target output word should be kept in the sample and gets updated, and we add to this a few (non-target) words as negative samples. "A probabilistic distribution is needed for the sampling process, and it can be arbitrarily chosen.... One can determine a good distribution empirically." (Rong 2014)

Milokov et al. also use a simple subsampling approach to counter the imbalance between rare and frequent words in the training set (for example, "in", "the", and "a" provide less information value than rare words). Each word in the training set is discarded with probability P(wi) where:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

Here, *f(wi)* is the frequency of word *wi* and *t* is a chosen threshold, typically around 10-5.

DEEP LEARNING IN NEURAL NETWORKS: AN OVERVIEW



What a wonderful treasure trove is Jürgen Schmidhuber's "Deep Learning in Neural Networks: An Overview" (2014)! It provides all the background you need to gain an overview of deep learning (DL) and how we got there (as of 2014) through the preceding decades.

Schmidhuber 2014

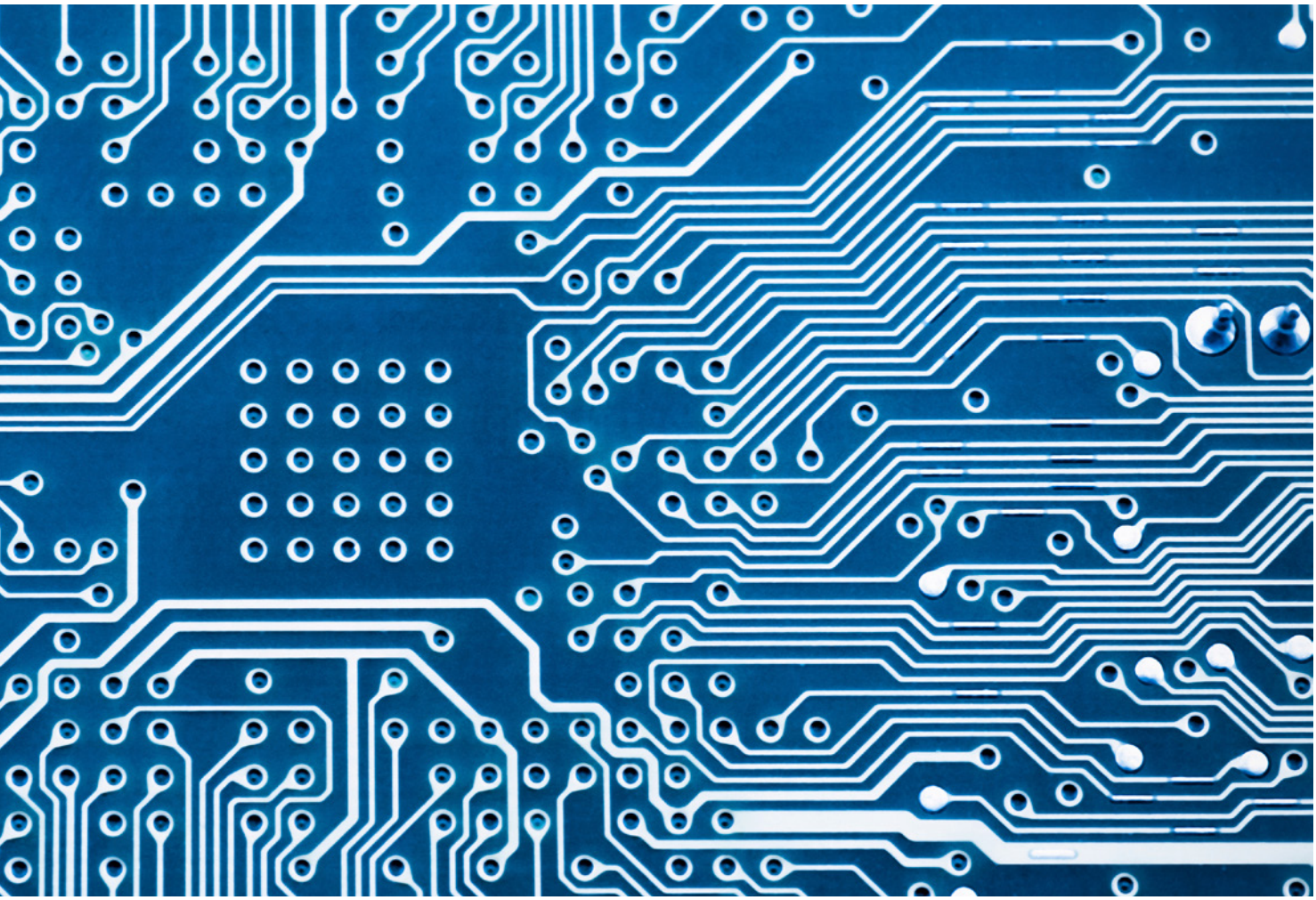
The paper runs 35 pages and then there are 53 pages of references. As a rough guess, that's somewhere around 900 referenced works. Now, I know that many of you think I read a lot of papers — just over 200 a year — but if I did nothing but review these key works in the development of deep learning, it would take me about four and a half years to get through them at that rate! And when I'd finished, I'd still be about six years behind the new current state of the art! It's a good reminder of how vast is the computer-science body of knowledge that we've built up over the last half-century.

I shall now attempt to condense a 35-page summary of 900 papers into a single chapter! Needless to say, there's a *lot* more detail in the full paper and references than I can cover here. We'll look at the following topics: credit assignment paths and the question of how deep is deep, key themes of deep learning, developments in supervised and unsupervised learning methods, reinforcement learning, and a short look at where things might be heading.

How deep is deep?

We don't know (but we'll see that "10" is *very deep*...).

Unsupervised learning can facilitate both supervised and reinforcement learning by first encoding essential features of inputs in a way that describes the original data in a less redundant or more compact way. These codes become the new inputs for supervised or reinforcement learning.



Which modifiable components of a learning system are responsible for its success or failure? What changes to them help improve performance? This has been called the fundamental credit assignment problem (Minsky 1963).... The present survey will focus on the narrower, but now commercially important, subfield of deep learning (DL) in artificial neural networks (NNs).... Learning or credit assignment is about finding weights that make the NN exhibit desired behaviour — such as driving a car. Depending on the problem and how the neurons are connected, such behaviour may require long causal chains of computational stages, where each stage transforms (often in a non-linear way) the aggregate activation of the network. Deep learning is about accurately assigning credit across many such stages. ***(All quotes from Schmidhuber 2014.)***

Feed-forward neural networks (FNNs) are acyclic; *recurrent* neural networks (RNNs) are cyclic. Schmidhuber wrote, "In a sense, RNNs are the deepest of all NNs," as in principle they can create and process memories of arbitrary sequences of input patterns.

To measure whether credit assignment in a given NN application is of the deep or shallow type, I introduce the concept of credit assignment paths or CAPs, which are chains of possibly causal links between events... e.g. from input through hidden to output layers in FNNs, or through transformations over time in RNNs.

If a credit assignment path (a path through the graph starting with an input) is of the form (..., k, t, ..., q), where k and t are the first successive elements with *modifiable weights* (it's possible that t = q), then the length of the suffix list t...q is the path's *depth*.

This depth limits how far backwards credit assignment can move down the causal chain to find a modifiable weight.... The depth of the deepest CAP within an event sequence is called the solution depth.... Given some fixed NN topology, the smallest depth of any solution is called the problem depth. Sometimes we also speak of the depth of an architecture: supervised learning FNNs with fixed topology imply a problem-independent maximal problem depth bounded by the number of non-input layers.... In general, RNNs may learn to solve problems of potentially unlimited depth.

So where does *shallow learning* end, and truly deep learning begin? "Discussions with DL experts have not yet yielded a conclusive response to this question!" wrote Schmidhuber.

Instead of committing myself to a precise answer, let me just define for the purposes of this overview: problems of depth > 10 require Very Deep Learning.

Key themes

Several themes recur across the different types of deep learning:

Dynamic programming can help to facilitate credit assignment. In supervised learning, back-propagation (BP) itself can be viewed as a method derived from dynamic programming. Dynamic programming can also help to reduce problem depth in traditional reinforcement learning, and dynamic programming algorithms are essential for systems that combine concepts of NNs and graphical models, such as hidden Markov models (HMMs).

> **GPUs excel at the fast matrix and vector multiplications required for NN training, where they can speed up learning by a factor of 50 or more.**

Unsupervised learning can facilitate both supervised and reinforcement learning by first encoding essential features of inputs in a way that describes the original data in a less redundant or more compact way. These codes become the new inputs for supervised or reinforcement learning.

Many methods learn *hierarchies* of more and more abstract data representations — continuously learning concepts by combining previously learned concepts.

"In the NN case, the *minimum description length* principle suggest that a low NN weight complexity corresponds to high NN probability in the Bayesian view, and to high generalization perfor-

mance, without over-fitting the training data. Many methods have been proposed for *regularizing* NNs — that is, searching for solution-computing but simple, low-complexity supervised learning NNs."

GPUs! GPUs excel at the fast matrix and vector multiplications required for NN training, where they can speed up learning by a factor of 50 or more.

Supervised and unsupervised learning

Supervised learning assumes that input events are independent of earlier output events (which may affect the environment through actions causing subsequent perceptions). This assumption does not hold in the broader fields of sequential decision making and reinforcement learning.

Early supervised NNs were essentially variants of linear regression methods going back to at least the early 1800s! In 1979, the *neocognitron* was the first artificial NN that deserved the attribute "*deep*", based on neurophysiological insights from studies of the visual cortex of cats carried out in the 1960s.

It introduced convolutional NNs (today often called CNNs or convnets), where the (typically rectangular) receptive field of a convolutional unit with a given weight vector (a filter) is shifted step by step across a two-dimensional array of input values, such as the pixels of an image (usually there are several such filters). The resulting 2-D array of subsequent activation events of this unit can then provide inputs to higher-level units, and so on.... The neocognitron is very similar to the architecture of modern, contest-winning, purely supervised, feed-forward, gradient-based deep learners with alternating convolutional and down-sampling layers.

The neocognitron did not use back-propagation. The first neural-network-specific application of efficient back-propagation was described in 1981. For weight-sharing FNNs or RNNs with activation spreading through some differentiable function f_t, a single iteration of gradient descent through back-propagation computes changes of all the weights w_i. Forward and backward passes are reiterated until sufficient performance is reached.

The back-propagation algorithm itself is wonderfully simple. In the description below, d_t is the target output value and *v(t,k)* is a function that gives the index of the weight that connects t and k. Each weight w_i is associated with a real-valued variable Δ_i initialized to 0.

Alg. 5.5.1: One iteration of BP for weight-sharing FNNs or RNNs

for $t = T, \ldots, 1$ **do**
to compute $\frac{\partial E}{\partial net_t}$, inititalize real-valued error signal variable δ_t by 0;
if x_t is an input event then continue with next iteration;
if there is an error e_t then $\delta_t := x_t - d_t$;
add to δ_t the value $\sum_{k \in out_t} w_{v(t,k)} \delta_k$; *(this is the elegant and efficient recursive chain rule application collecting impacts of net_t on future events)*
multiply δ_t by $f_t'(net_t)$;
for all $k \in in_t$ add to $\Delta_{w_{v(k,t)}}$ the value $x_k \delta_t$
end for
change each w_i in proportion to Δ_i and a small real-valued learning rate

(Schmidhuber 2014)

As of 2014, this simple BP method is still the central learning algorithm for FNNs and RNNs. Notably, most contest-winning NNs up to 2014 did not augment supervised BP by some sort of unsupervised learning....

In 1989, back-propagation was combined with neocognitron-like weight-sharing convolutional neural layers with adaptive connections....

This combination, augmented by max-pooling, and sped up on graphics cards, has become an essential ingredient of many modern, competition-winning, feed-forward, visual deep learners. This work also introduced the MNIST data set of handwritten digits, which over time has become perhaps the most famous benchmark of machine learning.

In max-pooling, a two-dimensional layer or array of unit activations is partitioned into smaller rectangular arrays. Each is replaced in a down-sampling layer by the activation of its maximally active unit.

Through the 1980s, it seemed that although BP allowed for deep problems in principle, it only worked for *shallow* problems. The reason for this was only fully understood in 1991, in Schmidhuber's student Sepp Hochreiter's diploma Ph.D. thesis. Schmidhuber comments:

Typical deep NNs suffer from the now famous problem of vanishing or exploding gradients. With standard activation functions, cumulative back-propagated error signals either shrink rapidly or grow out of bounds. In fact, they decay exponentially in the number of layers or CAP depth, or they explode. This is also known as the long time-lag problem.

This is the *fundamental deep-learning problem*, and several ways of partially overcoming it have been explored over the years:

Unsupervised pre-training can facilitate subsequent supervised credit assignment through back-propagation.

"L*ong short-term memory* (LSTM)"-like networks (described shortly) alleviate the problem though a special architecture unaffected by it.

Use of GPUs with more computing power allows the propagation of errors a few layers further down within reasonable time, even in traditional NNs. Schmidhuber notes, "That is basically what is winning many of the image-recognition competitions now, although this does not really overcome the problem in a fundamental way."

Hessian-free optimization (an advanced form of gradient descent) can alleviate the problem for FNNs and RNNs.

The space of weight matrices can be searched without relying on error gradients at all, thus avoiding the problem. *Random weight guessing sometimes works better than more sophisticated methods!* Other alternatives include *universal search* and the use of linear methods.

A working very deep learner of 1991 could perform credit assignment across hundreds of nonlinear operators or neural layers by using unsupervised pre-training for a hierarchy of RNNs. The basic idea is still relevant today. Each RNN is trained for a while in unsupervised fashion to predict its next input. From then on, only unexpected inputs (errors) con-

vey new information and get fed to the next higher RNN, which thus ticks on a slower, self-organising time scale. It can easily be shown that no information gets lost. It just gets compressed....

The supervised LSTM RNN could eventually perform similar feats to this deep RNN hierarchy without needing any unsupervised pre-training.

The basic LSTM idea is very simple. Some of the units are called constant error carousels (CECs). Each CEC uses as an activation function f, the identity function, and has a connection to itself with a fixed weight of 1.0. Due to f's constant derivative of 1.0 (d/dx f(x) = x is 1), errors back-propagated through a CEC cannot vanish or explode but stay as they are unless they "flow out" of the CEC to other, typically adaptive parts of the NN). CECs are connected to several nonlinear adaptive units (some with multiplicative activation functions) needed for learning nonlinear behavior.... CECs are the main reason why LSTM nets can learn to discover the importance of (and memorize) events that happened thousands of discrete time steps ago, while previous RNNs already failed in case of minimal time lags of 10 steps.

Hidden Markov models combined with FNNs dominated speech recognition up until the early 2000s. But when trained from scratch, LSTM obtained results comparable to these systems. By 2007, LSTM was outperforming them. "Recently, LSTM RNN/HMM hybrids obtained the best-known performance on medium-vocabulary and large-vocabulary speech recognition." LSTM RNNs have also won several international pattern-recognition competitions and have set numerous benchmark records on large and complex data sets.

Many competition-winning deep-learning systems today are either *stacks* of LSTM RNNs trained using *connectionist temporal classification* (CTC) or GPU-based max-pooling CNNs (GPU-MPCNNs). CTC is a gradient-based method for finding RNN weights that maximizes the probability of teacher-given label sequences, given (typically much longer and higher-dimensional) streams of real-valued input vectors.

An ensemble of GPU-MPCNNs was the first system to achieve superhuman visual pattern recognition in a controlled competition, namely, the IJCNN 2011 traffic-sign recognition contest in San Jose.... The GPU-MPCNN ensemble obtained 0.56% error rate and was twice better than human test subjects, three times better that the closest artificial NN competitor, and six times better than the best non-neural method.

An ensemble of GPU-MPCNNs also was the first to achieve human-competitive performance (around 0.2%) on MNIST in 2012. In the same year, GPU-MPCNNs achieved the best results on the ImageNet classification benchmark, and in a visual object detection contest — the ICPR 2012 contest on mitosis detection in breast-cancer histological images.

Such biomedical applications may turn out to be among the most important applications of DL. The world spends over 10% of GHP on healthcare (> 6 trillion USD per year), much of it on medical diagnosis through expensive experts. Partial automation of this could not only save lots of money, but also make expert diagnostics accessible to many who currently cannot afford it.

Reinforcement learning (RL)

Without a teacher, solely from occasional real-valued pain and pleasure signals, RL agents must discover how to interact with a dynamic, initially unknown environment to maximize their expected cumulative reward signals. There may be arbitrary, a priori unknown delays between actions and perceivable consequences. The problem is as hard as any problem of computer science, since any task with a computable description can be formulated in the RL framework.

In the general case, RL implies deep CAPs, but under the simplifying assumption of *Markov decision processes* (MDPs), the CAP depth can be greatly reduced. In an MDP, the current input of the RL agent conveys all information necessary to compute an optimal next output event or decision.

Perhaps the most well known RL NN is the world-class RL backgammon player (Tesauro, 1994), which achieved the level of human world champions by playing against itself... More recently, a rather deep GPU-CNN was used in a traditional RL framework to play several Atari 2600 computer games directly from 84×84-pixel 60-Hz video input.... Even better results are achieved by using (slow) Monte Carlo tree planning to train comparatively fast deep NNs.

For many situations the MDP assumption is unrealistic: "However, memories of previous events can help to deal with partially observable Markov decision problems (POMDPs)."

See also the work on neural Turing machines and memory networks from Google and Facebook respectively that was published after the Schmidhuber paper we're looking at.

Not quite as universal... yet both more practical and more general than most traditional RL algorithms are methods for direct policy search (DS). Without a need for value functions or Markovian assumptions, the weights of an FNN or RNN are directly evaluated on the given RL problem. The results of successive trials inform further search for better weights. Unlike with RL supported by BP, CAP depth is not a crucial issue. DS may solve the credit assignment problem without backtracking through deep causal chains of modifiable parameters — it neither cares for their existence, nor tries to exploit them.

The most general type of RL is constrained only by the fundamental limitations of computability. "Remarkably, there exist blueprints of *universal problem solvers* or *universal RL machines* for unlimited problem depth that are time-optimal in various theoretical senses." These can solve any well-defined problem as quickly as the unknown fastest way of solving it, save for an additive constant overhead that becomes negligible as the problem size grows.

Note that most problems are large; only few are small. AI and DL researchers are still in business because many are interested in problems so small that it is worth trying to reduce the overhead through less general methods, including heuristics.

Where next? (c. 2014)

...Humans learn to actively perceive patterns by sequentially directing attention to relevant parts of the available data. Near-future deep NNs will do so too, extending previous work since 1990 on NNs that learn selective attention through RL of (a) motor actions such as saccade control, and (b) internal actions controlling spotlights of attention within RNNs, thus closing the general sensorimotor loop through both external and internal feedback.

See the Google Deep Mind neural Turing machine paper for an example of a system with a memory and trainable attention mechanism.

Many recent DL results profit from GPU-based traditional deep NNs. Current GPUs, however, are little ovens, much hungrier for energy than biological brains, whose neurons communicate by brief spikes and often remain quiet. Many computational models of such spiking neurons have been proposed and analyzed.... Future energy-efficient hardware for DL in NNs may implement aspects of such models.

Schmidhuber reserves the last word for universal problem solvers:

The more distant future may belong to general-purpose learning algorithms that improve themselves in provably optimal ways, but these are not yet practical or commercially relevant.

HOW TO BUILD STATIC CHECKING SYSTEMS USING ORDERS OF MAGNITUDE LESS CODE

You start with something simple. Over time, things grow more and more complex and before you know it, it's hard to know what's going on. Brown et al.'s 2016 paper, "How to build static checking systems using orders of magnitude less code", is a delightful reminder of the power of stripping back to the essence, and just how powerful the simple approach can sometimes be. The context is the creation of static checkers for finding bugs in code.

Brown et al., ASPLOS '16

Research on bug finding has exploded in the past couple of decades. While researchers have explored many approaches, the dominant, near-universal trend has been towards increasing complexity.... the more complex a tool is; the worse it scales; the harder it is to debug and understand; and the more assumptions it makes, limiting the programs it can check. ***(All quotes from Brown et al. 2016.)***

The authors even had trouble understanding their own tools.

This paper is our reaction to these (and other) bitter experiences, and to the metastasizing complexity throughout the field. We have taken the simplest bug finding approach — static analysis — and further reduced its complexity by about two orders of magnitude.

The dramatically simpler result turns out to also be very effective.

Our empirical result that checkers can ignore the bulk of a programming language and yet be effective is viewed as "surprising" (to pick the most diplomatic word) by nearly everyone in the programming-language community we have discussed it with.

How is this combination of simplicity and effectiveness achieved? By ignoring most of the accidental complexity in language grammars (from the perspective of a bug-checking tool) and focusing only on the essen-

Whereas a traditional parser returns an error when it fails to parse a program, a micro-grammar-based parser simply slides forward by one token on non-matching input and tries again.

tial complexity that needs to be addressed for the problem at hand. Micro-parsers are created based on micro-grammars: partial grammars designed to extract checker-specific features only, and that don't need to understand the full programming-language grammar. The parsers are often less than 200 lines of code, and the checkers built on top of them often less than 40 lines of code. The authors are able to find over 700 bugs in widely used systems written in different languages, including bugs not found by commercial tools. In the spirit of COST ("configuration that outperforms a single thread"), we're going to have to learn how to COMPare ("checker that outperforms a micro-parser") code checkers.

The micro-parsing idea is easy to understand (which is kind of the point). It works as follows:

Whereas a traditional parser returns an error when it fails to parse a program, a micro-grammar-based parser simply slides forward by one token on non-matching input and tries again. Thus it matches all constructs described by the micro-grammar and skips past anything that doesn't match.

Within a grammar rule, developers can use wildcard non-terminals that lazily match any input up to a suffix.

For example, we can write a micro-grammar that matches if statements but does not care about the details of expressions: "S → if (wildcard)". When applied to a file with five if statements, the parser returns five parse trees. While a traditional parser creates a tree with (roughly) one node for every token, our parser returns a tree where lists of tokens match to each wildcard node. For example, parsing "if (x == y)" with the if statement micro-grammar would result in a normal if node and a wildcard node containing "["x", "==", "y"]".

The parsers and checkers are so simple that they can often be created in less than a day or even a couple of hours. And often parts of languages overlap enough that they can share almost the same micro-grammar.

Developers then build parsers recursively by defining a parser for each non-terminal and composing these parsers together to accept a micro-grammar. Adding a new non-terminal to a grammar equates to combining its parser with an existing one. For example, to create a parser for C's control flow, we first write parsers for if statements, while loops, for loops, etc. Each of these small parsers is around six lines of

Despite ignoring most of the language, our checkers are not limited to checking simple syntactic properties. We reimplement two flow-sensitive checkers originally built using a traditional checking system, and show that despite being orders of magnitude simpler, they find a similar number of true bugs.

code, and we glue them together into a 156-line C control-flow parser.

The overall framework is called µchex, and provides a set of built-in functions for common checking operations. It is tuned for "belief-style" checkers, where beliefs are facts implied by code. For example, x/y implies a belief that y is non-zero. The micro-grammar approach is not limited to this style of checking though.

Comparison to traditional checkers

Despite ignoring most of the language, our checkers are not limited to checking simple syntactic properties. We re-implement two flow-sensitive checkers originally built using a traditional checking system, and show that despite being orders of magnitude simpler, they find a similar number of true bugs.

The authors recreate two flow-sensitive checkers from their previous work: a null-pointer checker and a deadlock checker. The null-pointer checker uses two parsers, one for control flow and one for expressions. The (reusable) control-flow parser is 156 lines of code. The expression parser is 29 lines of code. The null-pointer checker operates on the resulting tree. It finds 233 true bugs and 12 false positives in Linux (5% false positives). The original checker found 102 bugs and 4 false positives. The checker also finds 63 bugs and 3 false positives in Firefox.

The deadlock checker computes a set of beliefs about whether a program can block. It finds 124 bugs in the Linux and no false positives. The original version found 123 bugs.

Comparison to state-of-the-art commercial tools

Micro-grammar checkers are not weak, and can help debug more sophisticated tools. When we check the same property as a state-of-the-art commercial tool and compare our results on the same code base, we find a roughly comparable number of bugs; we also find 190 bugs that the commercial tool misses. As a result, we think that our approach may be used as a safety net for more complex tools.

Brown et al. identify the commercial tool simply as "SystemX": "The company behind the tool has been developing static checkers for more than a decade, with a team that currently numbers in the low hundreds." Testing against Linux 4.4-rc7, the micro-grammar based checker finds 44 of the 122 that SystemX reports. But it also finds 190 bugs that SystemX does not. Of the bugs that SystemX finds but the micro-checker does not (72 in total), 66 occur because SystemX uses inter-procedural analysis and alias tracking, which the micro-checker does not.

Based on our past experiences, we believe simple inter-procedural analysis can be readily added to µchex. µchex could also implement trivial alias tracking, but non-trivial alias tracking is challenging without a full understanding of the checked language.

Adaptability

We repeatedly show that adopting an existing checker in our system to a new, previously unchecked language is a matter of writing a few lines of code (even when going from C++ to JavaScript!). Furthermore, writing a new checker from scratch is often in the tens of lines. This ease is in contrast to traditional static checking systems, which, because of their detailed

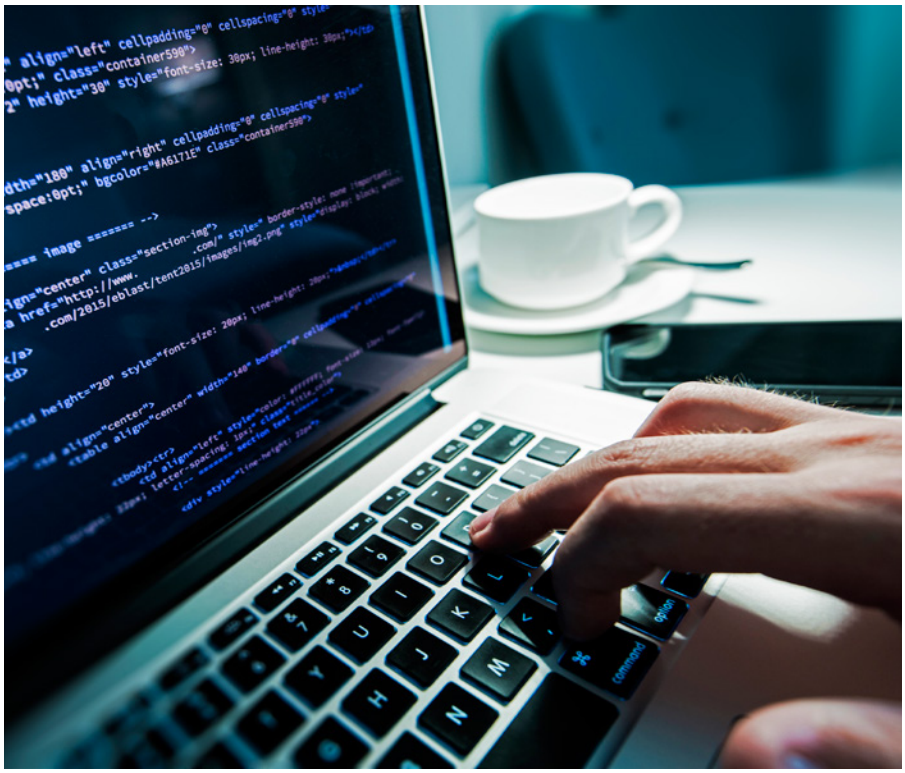
dependency on exact language semantics, might take months or years to retarget to a new language.

The authors took the null-pointer checker originally implemented in C and adapted it to look for null-pointer errors in C++, Java, and JavaScript. As they point out, creating or adapting a checker's front end for a new language can take months or years: "As one data point, it took us years to add Java checking to our C static tool, both in an academic and later in an industry setting."

Adapting the null pointer checker to C++ requires 27 lines of code, and finds 71 bugs. Even without any adaptation, the original tool finds 50 of those same bugs — a great demonstration of the power of being liberal in what you accept when the details don't matter. Adapting the checker to Java is a mere two lines of code: "the only real change is representing the dereference operator as "." instead of "->". Here's an example of an issue found by the checker in OpenJDK :

```
// implies output != null
int outputCapacity = output.
length - outputOffset;
int minOutSize = (decrypting?
  (estOutSize -
blockSize):estOutSize);
// implies output can be
null...
if ((output == null) ... )
...
```

Adapting the checker to JavaScript required 17 lines of code. It finds 25 bugs in Firefox and 7 in Node.js.



The authors then create new checkers from scratch for Dart and CPP (the C preprocessor). For CPP, a 40-line checker and parser finds 16 bugs in Firefox and seven bugs in Linux. For Dart, the authors implement four checkers based on common errors identified by active Dart developers: equals without hashcode, non-observable @Observable collections, spurious null checks, and incorrect length checks. Three of these checkers share the same 207-line Dart control-flow parser; the other one uses an independent seven-line parser.

The equals-without-hashcode checker is 28 lines and finds 13 bugs. The non-observable checker is 26 lines and finds two bugs. The null checker is nine lines. The incorrect length checker is seven lines and finds 152 bugs. The authors conclude that "Clearly,

even though these checkers are small and easy to build, they still find real-world errors that matter to developers."

Portability

One of our most unexpected results is that the same checker and parser combination can find bugs in many different languages. The intuition behind this fact is that if languages share a common heritage, some portion of their grammars may be similar, even though they may otherwise differ significantly. Micro-grammars can capture the overlapping portion while ignoring the rest of the grammar.

The same checker, unaltered, is run on C, C++, Java, and JavaScript code, looking for programming-logic errors in the form of tautological branching decisions — if statements with redundant branches, repeated conditions, and loop headers where the condition contradicts the increment "for (i = 0; i < len; i--)".

The redundant branches checker finds 48 true bugs and 12 false positives, plus eight additional instances in generated code in OpenJDK. The other checkers are also able to detect genuine bugs across all languages.

Summary of errors found using micro-grammars (Brown et al. 2016).

Checker	System	Bugs	False
Blocking while Locking	Linux (C)	124	0
Null Pointer	Linux (C)	233	12
	OpenJDK (Java)	42	3
	LLVM (C++)	10	1
	Firefox (C)	63	3
	Firefox (C++)	61	33
	Firefox (Java)	3	2
	Firefox (JavaScript)	25	6
	Node.js (JavaScript)	7	1
	All	444	61
Redundant Nested Macro Conditions	Linux (C macro)	7	0
	Firefox (C macro)	16	0
	All	23	0
Equals wo/ Hashcode	OpenJDK (Java)	73	2
	Dart SDK (Dart)	13	0
	All	93	2
Non-Observable Collection	Dart SDK (Dart)	2	0
Redundant Branches	Linux (C)	34	6
	OpenJDK (Java)	5 (8*)	1
	LLVM (C++)	2	0
	Firefox (C)	3	3
	Firefox (C++)	4	2
	All	48 (8*)	12
Redundant Conditions	Linux (C)	6	0
	OpenJDK (Java)	3	0
	Firefox (C++)	6	0
	LLVM (C++)	1	0
	All	16	0
Suspicious Loop Conditions	Linux (C)	5	2
	OpenJDK (Java)	2	0
	Firefox (C)	1	0
	Firefox (C++)	1	32‡
	LLVM (C++)	0	5
	Node.js (JavaScript)	2	0
	All	11	39
Total		761 (8*)	116
Spurious Null Check†	Dart SDK (Dart)	3	0
Incorrect Length Check†	Dart SDK (Dart)	152	0

* Generated code † Style and performance checkers
‡ Repeated patterns in related files

Table 1. Errors in all systems. We check Linux 4.4-rc7, Firefox 39.0, OpenJDK 8 b132, Node.js 0.12.7, LLVM 3.6.2 and Dart SDK 1.13.2 (including Core Libraries).

GORILLA

A FAST, SCALABLE, IN-MEMORY TIME-SERIES DATABASE

Error rates across one of Facebook's sites were spiking. The problem had first shown up a few minutes after it started through an automated alert triggered by an in-memory time-series database called Gorilla. One set of engineers mitigated the immediate issue. A second group set out to find the root cause. They fired up Facebook's time-series correlation engine built on top of Gorilla and searched for metrics showing a correlation with the errors. This showed that copying a release binary to Facebook's web servers (a routine event) caused an anomalous drop in memory used across the site.

Pelkonen et al. 2015

In the 18 months prior to Pelkonen et al. 2015 publishing "Gorilla: A fast, scalable, in-memory time series database", Gorilla helped Facebook engineers identify and debug several such production issues.

An important requirement to operating [these] large scale services is to accurately monitor the health and performance of the underlying system ad quickly identify and diagnose problems as they arise. Facebook uses a time series database to store system measuring data points and provides quick query functionalities on top. (All quotes from Pelkonen et al. 2015.)

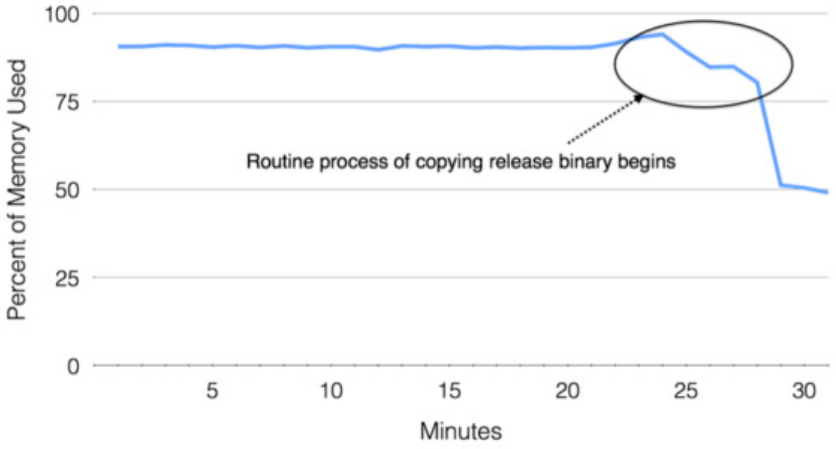


Figure 10: When searching for the root cause for a site-wide error rate increase, Gorilla's time series correlation found anomalous events that were correlated in time, namely a drop in memory used when copying a newly released binary.

(Pelkonen et al. 2015)

As of Spring 2015, Facebook's monitoring systems generated more than 2 billion unique time series of counters, with about 12 million data points added per second — *over 1 trillion data points per day*. Here then are the design goals for Gorilla:

- Store 2 billion unique time series, identifiable via a string key.
- Insert 700 million data points (time stamp and floating-point value) per minute.
- Retain data for fast querying over the last 26 hours.
- Allow up to 40,000 queries per second at peak.
- Reads must succeed in under 1 millisecond.
- Support time series with up to 15-second granularity (four points/minute/time series).
- Maintain two in-memory replicas (not co-located) for DR.
- Continue to serve reads in the face of server crashes.
- Support fast scans over all in-memory data.
- Handle continued growth in time-series data of double per year!

To meet the performance requirements, Gorilla is built as an *in-memory* TSDB that functions as a *write-through cache* for monitoring data ultimately written to an HBase data store. To meet the requirements to store 26 hours of data in memory, Gorilla incorporates a new time-series compression algorithm that achieves an average 12-times reduction in size. The in-memory data struc-

tures allow fast and efficient scans of all data while maintaining constant time lookup of individual time series.

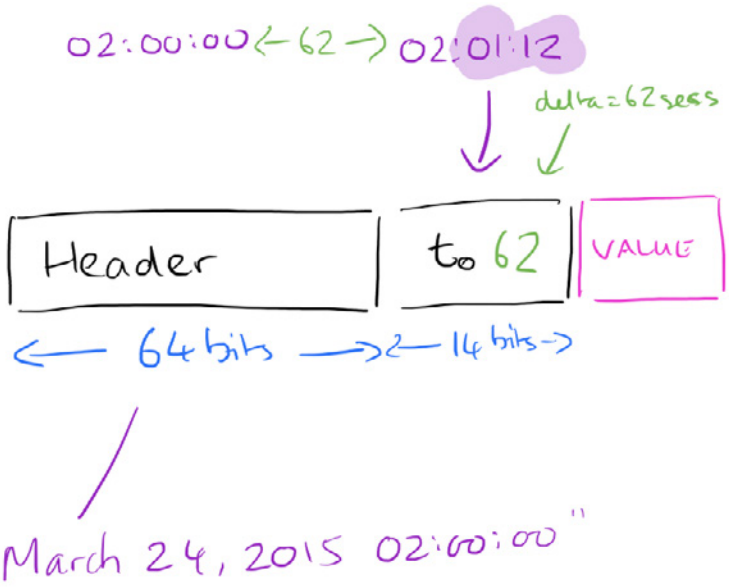
The key specified in the monitoring data is used to uniquely identify a time series. By sharding all monitoring data based on these unique string keys, each time-series dataset can be mapped to a single Gorilla host. Thus, we can scale Gorilla by simply adding new hosts and tuning the sharding function to map new time-series data to the expanded set of hosts. When Gorilla was launched to production 18 months ago, our dataset of all time-series data inserted in the past 26 hours fit into 1.3 TB of RAM evenly distributed across 20 machines. Since then, we have had to double the size of the clusters twice due to data growth, and are now running on 80 machines within each Gorilla cluster. This process was simple due to the share-nothing architecture and focus on horizontal scalability.

The in-memory data structure is anchored in a C++ Standard Library unordered map. This proved to have sufficient performance and no issues with lock contention. For persistence, Gorilla stores data in GlusterFS, a POSIX-compliant distributed file system with triple replication. Pelkonen et al. note that HDFS or other distributed file systems would have sufficed just as easily. For more details on the data structures and how Gorilla handles failures, see sections 4.3 and 4.4 in the paper. I want to focus here on the techniques Gorilla uses for time-series compression to fit all of that data into memory!

Time-series compression in Gorilla

Gorilla compresses data points within a time series with no additional compression used** across **time series. Each data point is a pair of 64-bit values representing the time stamp and value at that time. Time stamps and values are compressed separately using information about previous values.

When it comes to time stamps, a key observation is that most sources log points at fixed intervals (e.g. one point every 60 seconds). Every now and then, the data point may be logged a little bit early or late (e.g., a second or two), but this window is normally constrained. We're now entering a world where every *bit* counts, so if we can represent successive time stamps with very small numbers, we're winning.... Each data block is used to store two hours of data. The block header stores the starting time stamp, aligned to this two-hour window. The first time stamp in the block (first entry after the start of the two-hour window) is then stored as a delta from the block start time, using 14 bits. The 14 bits is enough to span a bit more than four hours at one-second resolution so we know we won't need more than that.



For all subsequent time stamps, we compare *deltas*. Suppose we have a block start time of 02:00:00, and the first time

stamp is 62 seconds later at 02:01:02. The next data point is at 02:02:02, 60 seconds later. Comparing these two deltas, the second delta (60 seconds), is two seconds shorter than the first one (62 seconds). So we record "-2". How many bits should we use to record the -2? Ideally as few as possible! We can use tag bits to tell us how many bits the actual value is encoded with. The scheme works as follows:

Calculate the *delta of deltas*: *D = (tn - tn-1) - (tn-1 - tn-2)*

Encode the value according to the following table:

D	tag bits	# value bits	total bits
0	0	0	1
[-63,64]	10	7	9
[-255,256]	110	9	12
[-2047,2048]	1110	12	16
>2048	1111	32	36

The particular values for the time ranges were selected by sampling a set of real time series from production systems and choosing the ranges that gave the best compression ratios.

This produces a data stream that looks like this:

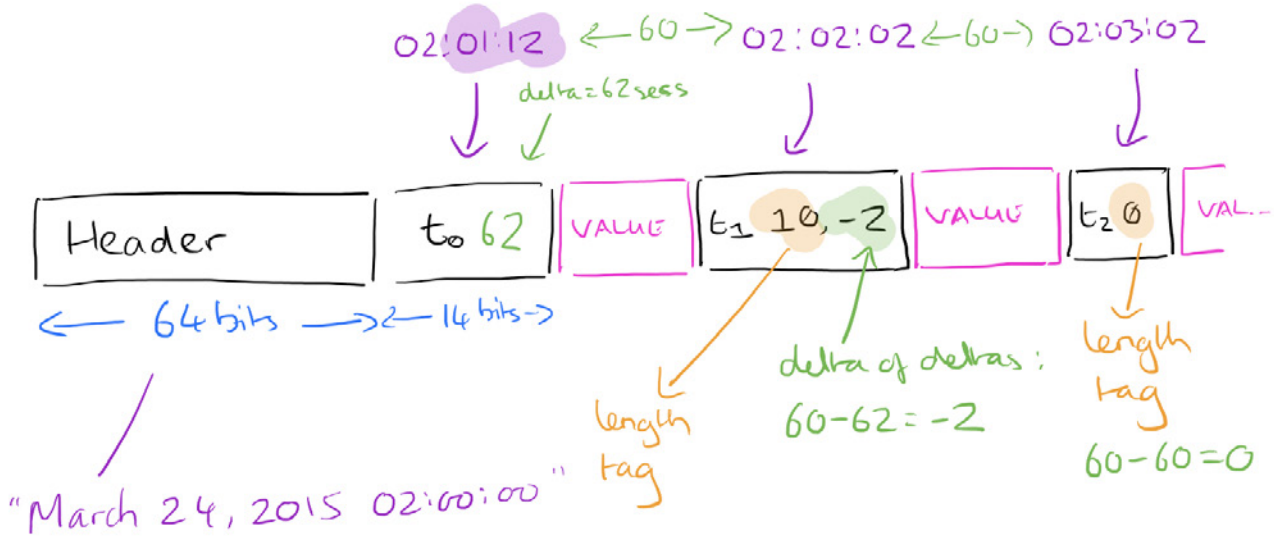


Figure 3 shows the results of time-stamp compression in Gorilla. We have found that about 96% of all time stamps can be compressed to a single bit.

(That is, 96% of all time stamps occur at regular intervals, such that the delta of deltas is zero).

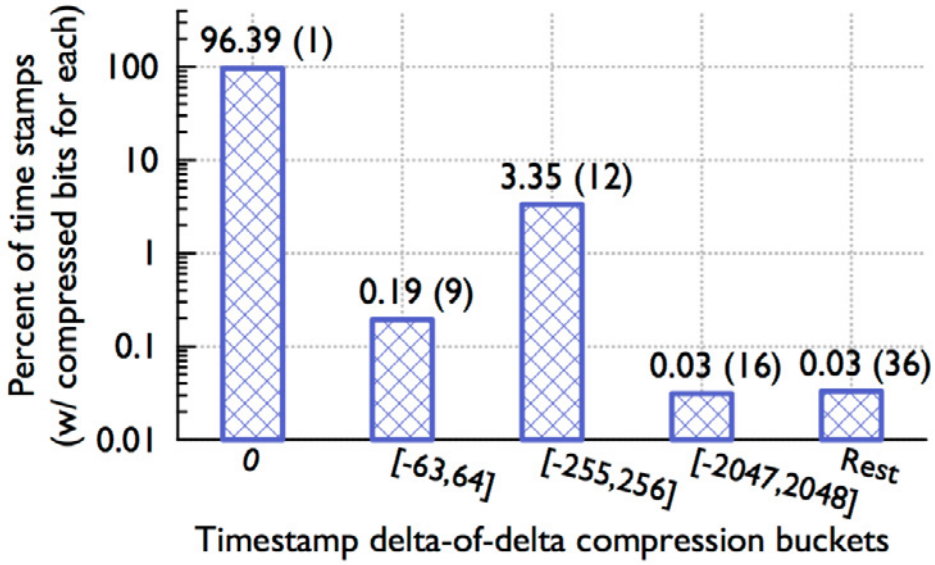


Figure 3: Distribution of time stamp compression across different ranged buckets. Taken from a sample of 440,000 real time stamps in Gorilla.

(Pelkonen et al. 2015)

So much for time stamps; what about the data values themselves?

We discovered that the value in most time series does not change significantly when compared to its neighboring data points. Further, many data sources only store integers. This allowed us to tune the expensive prediction scheme in [25] to a simpler implementation that merely compares the current value to the previous value. If values are close together (then) the sign, exponent, and first few bits of the mantissa will be identical. We leverage this to compute a simple XOR of the current and previous values rather than employing a delta encoding scheme.

The in-memory data structure is anchored in a C++ Standard Library unordered map. This proved to have sufficient performance and no issues with lock contention. For persistence, Gorilla stores data in GlusterFS, a POSIX-com pliant distributed file system with triple replication.

> **96% of all time stamps can be compressed to a single bit.**

Decimal	Double Representation	XOR with previous
12	0x4028000000000000	
24	0x4038000000000000	0x0010000000000000
15	0x402e000000000000	0x0016000000000000
12	0x4028000000000000	0x0006000000000000
35	0x4041800000000000	0x0069800000000000

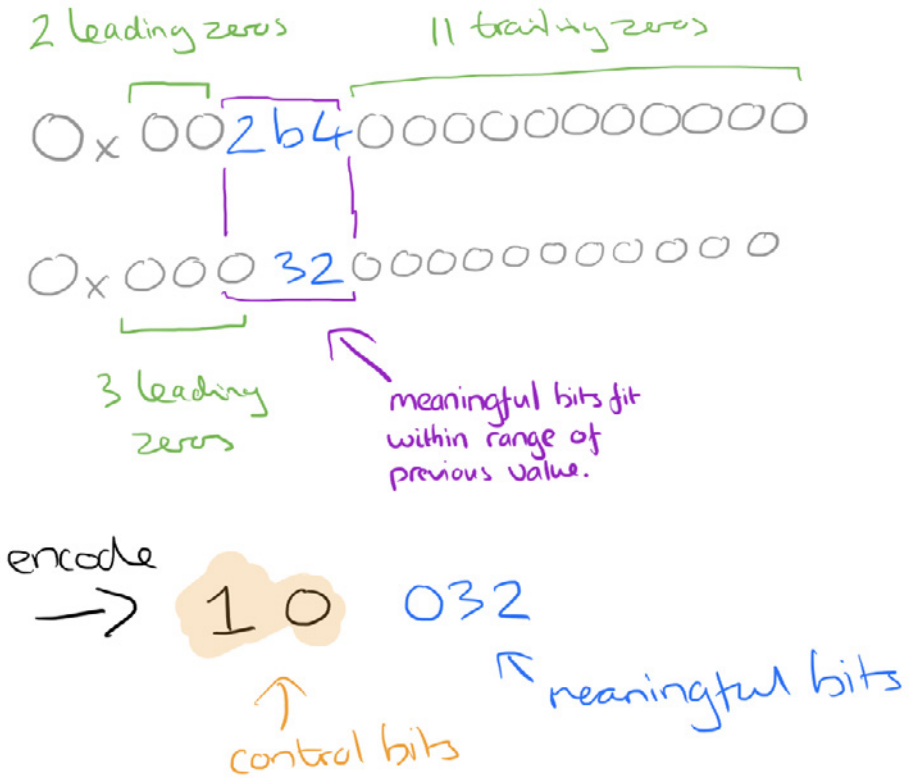
Decimal	Double Representation	XOR with previous
15.5	0x402f000000000000	
14.0625	0x402c200000000000	0x0003200000000000
3.25	0x400a000000000000	0x0026200000000000
8.625	0x4021400000000000	0x002b400000000000
13.1	0x402a333333333333	0x000b733333333333

Figure 4: Visualizing how XOR with the previous value often has leading and trailing zeros, and for many series, non-zero elements are clustered.

(Pelkonen et al. 2015)

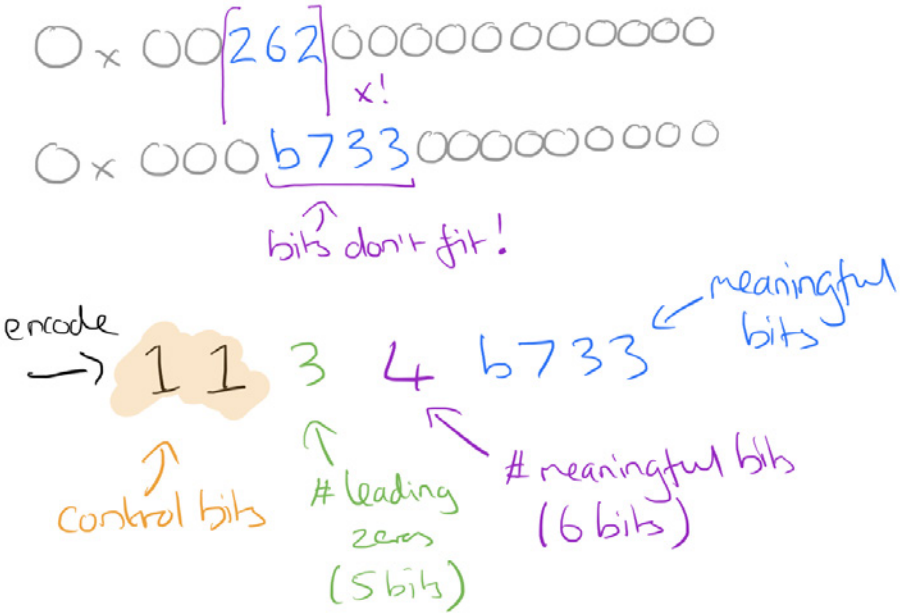
The values are then encoded as follows:

- The first value is stored with no compression.
- For all subsequent values, XOR with the previous value. If the XOR results in zero (i.e. the same value), store the single bit "0".
- If the XOR doesn't result in zero, it's still likely to have a number of leading and trailing zeros surrounding the meaningful bits. Count the number of leading zeros and the number of trailing zeros — e.g. 0x0003200000000000 has three leading zeros and 11 trailing zeros. Store the single bit "1" and then....
- If the previous stored value also had the same number or fewer leading zeros and the same number or fewer trailing zeros, we know that all the meaningful bits of the value we want to store fall within the meaningful bit range of the previous value:



Store the control bit "0" and then store the meaningful XOR'd value (i.e. 032) in the example above.

- If the meaningful bits do not fit within the meaningful bit range of the previous value, then store the control bit "1" followed by the number of leading zeros in the next five bits, the length of the meaningful XOR'd value in the next six bits, and finally the meaningful bits of the XOR'd value.



Roughly 51% of all values are compressed to a single bit since the current and previous values are identical. About 30% of the values are compressed with the control bits "10" with an average compressed size of 26.6 bits. The remaining 19% are compressed with control bits "11", with an average size of 36.9 bits, due to the extra overhead required to encode the length of leading zero bits and meaningful bits.

Building on top of Gorilla

Gorilla's low-latency processing (over 70 times faster than the system it replaced) enabled the Facebook team to build a number of tools on top; these include horizon charts, aggregated roll-ups that update based on all completed buckets every two hours, and a correlation engine that we saw being used in the opening case study.

The correlation engine calculates the Pearson product-moment correlation coefficient (PPMCC), which compares a test time series to a large set of time series. We find that PPMCC's ability to find correlation between similarly shaped time series, regardless of scale, greatly helps automate root-cause analysis and answer the question "What happened around the time my service broke?" We found that this approach gives satisfactory answers to our question and was simpler to implement than similarly focused approaches described in the literature [10, 18, 16]. To compute PPMCC, the test time series is distributed to each Gorilla host along with all of the time-series keys. Then, each host independently calculates the top N correlated time series, ordered by the absolute value of the PPMCC compared to the needle, and returning the time-series values. In the future, we hope that Gorilla enables more advanced data-mining techniques on our monitoring time-series data, such as those described in the literature for clustering and anomaly detection [10, 11, 16].

Summing up lessons learned, the authors provide three further takeaways:

1. Prioritize recent data over historical data. Why things are broken right now is a more pressing question than why they were broken two days ago.
2. Read latency matters — without this, the more advanced tools built on top would not have been practical.
3. High availability trumps resource efficiency.

We found that building a reliable, fault-tolerant system was the most time-consuming part of the project. While the team prototyped a high-performance, compressed, in-memory TSDB in a very short period of time, it took several more months of hard work to make it fault tolerant. However, the advantages of fault tolerance were visible when the system successfully survived both real and simulated failures.

A SURVEY OF AVAILABLE CORPORA FOR BUILDING DATA-DRIVEN DIALOGUE SYSTEMS

Bear with me — this is more interesting than it sounds. Yes, the 46-page "A survey of available corpora for building data-driven dialogue systems" (Serban et al. 2015) does include a catalogue of data sets with dialogues from different domains, but it also includes a high-level survey of techniques that are used in building dialogue systems (a.k.a. chatbots). In particular, it focuses on data-driven systems — i.e., those that incorporate some kind of learning from data.

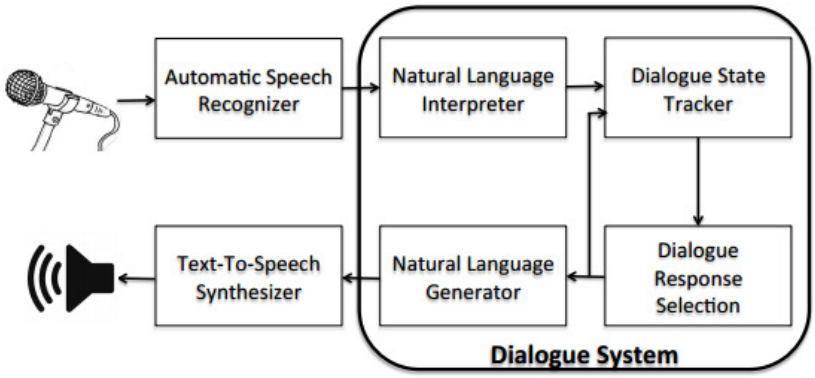
Serban et al. 2015

...A wide range of data driven machine learning methods have been shown to be effective in natural-language processing, including tasks relevant for dialogue such as dialogue-policy learning, dialogue-state tracking, and natural-language generation. (All quotes from Serban et al. 2015.)

This particular paper focuses on corpus-based learning, meaning we have been able to build up or have access to a data set on which we can train our models. If we want to build a defensible machine-learning-based business, having access to quality sources of data to which our competitors do not is a good start. Out of scope is training dialogue systems through live interaction with humans — but there are some references to follow up on this.

Anatomy of a dialogue system

The standard architecture for a dialogue system looks like this:



(Serban et al. 2015)

Natural-language interpretation and generation are core NLP problems with applications well beyond dialogue systems. For building chatbots, where we assume written input and output,

we can leave out the speech recogniser and synthesiser. I had naively assumed that if you had a good working system that can deal with textual inputs and outputs, it would be a simple matter of bolting a speech-to-text recogniser in front of the system in order to build a voice-driven assistant. It turns out it's not quite as simple as that, since the way we speak and the way we write have important differences.

The distinction between spoken and written dialogues is important, since the distribution of utterances changes dramatically according to the nature of the interaction.... Spoken dialogues tend to be more colloquial, use shorter words and phrases, and are generally less well formed, as the user is speaking in a train-of-thought manner. Conversely, in written communication, users have the ability to reflect on what they are writing before they send a message. Written dialogues can also contain spelling errors or abbreviations, which are generally not transcribed in spoken dialogues.

Even written dialogue — e.g. for movies and plays and in novels – has apparent distinctions from real speech, which leads to Serban et al's wonderful observation that "Nevertheless, recent studies have found that spoken language in movies resembles human spoken language." As an occasional movie watcher, I had never thought to question that, or that a study might be necessary to demonstrate it!

Anyway, I digress. Within dialogue systems, we can distinguish between goal-driven systems — such as travel assistants or technical-support services — that aim to accomplish some goal or task and non-goal-driven systems such as language-learning tools or computer-game characters. Most startups building chatbots will be building goal-driven systems.

Initial work on goal-driven dialogue systems primarily used rule-based systems... with the distinction that machine-learning techniques have been heavily used to classify the intention (or need) of the user, as well as to bridge the gap between text and speech. Research in this area started to take off during the mid '90s, when researchers began to formulate dialogue as a sequential-decision-making problem based on Markov decision processes.

Commercial systems to date are *highly domain specific* and

heavily based on *handcrafted features*. "In particular, *the datasets are usually constrained to a very small task*," observe Serban et al.

Discriminative models and supervised learning

Discriminative models, which use supervised learning to predict labels, can be used in many parts of a dialogue system — e.g., to predict the intent of a user in a dialogue, conditioned on what they have said. Here the intent is the label, and the conditioned utterances are called *conditioning variables* or *inputs*.

Discriminative models can be similarly applied in all parts of the dialogue system, including speech recognition, natural language understanding, state tracking, and response selection.

One popular approach is to learn a probabilistic model of the labels; another is to use maximum-margin classifiers such as support-vector machines. Discriminative models may be trained independently and then plugged in to fully deployed dialogue systems.

Answering back

When it comes to choosing what your chatbot is going to say (e.g., in response to a user message), there are again two broad distinctions. The simpler approach is to select deterministically from a fixed set of possible responses (which may of course use parameter substitution):

The model maps the output of the dialogue tracker or natural-language understanding modules together with the dialogue history (e.g. previous tracker outputs and previous system actions) and external knowledge (e.g. a database, which can be queried by the system) to a response action.

> **"The distinction between spoken and written dialogues is important, since the distribution of utterances changes dramatically according to the nature of the interaction.**

This approach effectively bypasses the natural-language-generation part of the system. The system designers may have crafted the fixed responses up front, but there are also systems that effectively search through a database of dialogues and pick the responses that have the most similar context:

...The dialogue history and tracker outputs are usually projected into a Euclidean space (e.g. using TF-IDF bag-of-words representations) and a desirable response region is found (e.g. a point in the Euclidian space). The optimal response is then found by projecting all potential responses into the same Euclidean space, and the response closest to the desirable response region is selected.

More complex chatbots generate their own responses. Using a method known as "beam search" they can generate highly probably responses. The approach is similar to that used in a paper on sequence-to-sequence machine translation that I recently looked at. Short (single request/response) conversations are simpler than those that need to be able to handle multiple interactive turns.

For example, an interactive system might require steps of clarification from the user before being able to offer pertinent information. Indeed, this is a common scenario: many dialogues between humans, as well as between humans and machines, yield significant ambiguity, which can usually be resolved over the course of the dialogue. This phase is sometimes referred to as the grounding process. To tackle such behaviors, it is crucial to have access to dialogue corpora with long interactions, which include clarifications and confirmations, which are ubiquitous in human conversations. The need for such long-term interactions is confirmed by recent empirical results, which show that longer interactions help generate appropriate responses.

Incorporating external knowledge

Chatbots may rely on more than just dialogue corpora for training. When building a goal-driven dialogue system for movies, Dodge et al. ("Evaluating prerequisite qualities for learning end-to-end dialog systems") identify four tasks that such a working dialogue system should be able to perform: answering questions, recommendation, answering questions with recommendation, and casual conversation. They use four different subsets of data to train models for these tasks: a QA dataset from the Open Movie Database (OMDb) of 116,000 examples with accompanying movie and actor metadata in the form of knowledge triples, a recommendation dataset from MovieLens with 110,000 users and 1 million questions, a combined recommendation and QA dataset with 1 million conversations of six turns each; and a discussion dataset from Reddit's movies subreddit.

The use of external information is usually of great importance to dialogue systems, especially goal-driven ones. This could include structured information — such as bus or train timetables for answering questions about public transport — typically contained in relational databases or similar collections. It's also possible to take advantage of structured external knowledge from general natural-language processing databases and tools. Some good sources include:

- WordNet, with lexical relationships between words for over a thousand words;
- VerbNet, with lexical relationships between verbs; and
- FrameNet, which contains "word senses" for over 10,000 words.

Tools include parts-of-speech taggers, word-category classifiers, word-embedding models, named-entity recognition models, semantic-role labelling models, semantic-similarity models, and sentiment-analysis models.

If you're building a new application and don't have ready access to large corpora for training, you may also be able to *transfer* learning from related datasets to bootstrap the learning process. As we saw in the previous chapter on Deep Learning In Neural Networks Serban et al. note a reference for use of related data sets in pre-training a model as an effective method of scaling up to complex environments in several branches of machine learning and particularly in deep learning.

An example of this approach in action is the work of Forgues et al. on *dialogue act classification* (classify a user's utterance as one out of *a selection of* dialogue acts).

They created an utterance-level representation by combining the word embeddings of each word, for example, by summing the word embeddings or taking the maximum w.r.t. each dimension. These utterance-level representations, together with word counts, were then given as inputs to a linear classifier to classify the dialogue acts. Thus, Forgues et al. showed that by leveraging another, substantially larger, corpus they were able to improve performance on their original task.

What were you saying?

Tracking the state of a conversation is a whole sub-genre of its own, which goes by the name of *dialogue-state-tracking* challenge (DSTC). It is framed as a classification problem: given current input to the dialogue-state tracker plus any relevant external knowledge from other sources (e.g., the timetable information from our previous example), the goal is to output a probability distribution over a set of predefined hypotheses, plus a special "REST hypothesis" that captures the probability that none of the others are correct. For example, the system may believe with high confidence that the user has requested timetable information for the current day. DSTC models include both statistical approaches and handcrafted systems.

More sophisticated models take a dynamic Bayesian approach by modeling the latent dialogue state and observed tracker outputs in a directed graphical model.... Non-Bayesian data-driven models have also been proposed.

Longer-term memories

We recently looked at memory networks and neural Turing machines that can store some part of their input in memory and use this to perform a variety of tasks.

Although none of these models are explicitly designed to address dialogue problems, the extension by Kumar et al. to dynamic memory networks specifically differentiates between episodic and semantic memory. In this case, the episodic memory is the same as the memory used in the traditional memory networks paper, which is extracted from the input, while the semantic memory refers to knowledge sources that are fixed for all inputs. The model is shown to work for a variety of NLP tasks, and it is not difficult to envision an application to dialogue utterance generation where the semantic memory is the desired external knowledge source.

Personality

As if all of the above wasn't hard enough, you may also want your bot to exhibit some kind of consistent personality.

Thus, attaining human-level performance with dialogue agents may well require personalization, i.e. models that are aware and capable of adapting to their intelocutor. Such capabilities could increase the effectiveness and naturalness of generated dialogues. We see personalization of dialogue systems as an important task, which so far has been mostly untouched.

The last word

There's plenty more detail and several additional topics in the original paper, which I have skipped over. If this topic interests you, it's well worth checking out. I'll leave you with the authors' closing thought:

There is strong evidence that over the next few years, dialogue research will quickly move towards large-scale data-driven model approaches, in particular in the form of end-to-end trainable systems as is the case for other language-related applications such as speech recognition, machine translation and information retrieval.... While in many domains, data scarcity poses important challenges, several potential extensions, such as transfer learning and incorporation of external knowledge, may provide scalable solutions.

HERE IS WHAT WE'VE COVERED IN THE PREVIOUS ISSUE

ARIES

A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks

NO COMPROMISES

Distributed Transactions with Consistency, Availability, and Performance

HYPERLOGLOG IN PRACTICE

Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm

ALL FILE SYSTEMS ARE NOT CREATED EQUAL

On the Complexity of Crafting Crash Consistent Applications

NOT-QUITE-SO-BROKEN TLS

Lessons in re-engineering a security protocol specification and implementation