September 2018 COMPUTER SCIENCE APPLIED Issue #8

THE MORNING PAPER
QUARTERLY REVIEW

DEEP-LEARNING SCALING IS PREDICTABLE, EMPIRICALLY

COMPUTING MACHINERY AND INTELLIGENCE

DEEPTEST: AUTOMATED TESTING OF DNN-DRIVEN AUTONOMOUS CARS

Lessons in Machine Learning from Alan Turing, Google's Deep Mind, and Baidu, plus testing self-driving cars, and searching for code.

InfoQueue

CONTENTS

Issue #8, September 2018

QCon.ai by InfoQ

April 15-17, 2019
San Francisco

APPLIED AI AND ML CONFERENCE

for software engineers

*rather than data scientists

Why should I attend?

Learn how software innovators are applying AI & machine learning in **use-case oriented sessions**

Hear about the **tools and techniques** of AI & machine learning

Go in-depth on key topics in our **1 day of workshops**

Meet with AI and ML leaders from innovator and early adopter companies

Gain valuable insights and ideas to shape your AI and machine learning projects

Learn how innovator companies are **applying AI & machine learning in their businesses**

Find out more!

WELCOME...

Adrian Colyer

Welcome to this AI-themed edition of The Morning Paper Quarterly. I've selected five paper write-ups which first appeared on The Morning Paper blog over the last year.

To kick things off, we're going all the way back to 1950! Alan Turing's paper "Computing Machinery and Intelligence" is a classic that gave us the Turing test, but also so much more. In it, Turing puts forward the idea that instead of directly building a computer with the sophistication of a human adult mind, we should break the problem down into two parts: building a simpler *child* program with the capability to learn and building an *education process* through which the child program can be taught. Writing almost 70 years ago, Turing expresses the hope that machines will eventually compete with men in all purely intellectual fields. But where should we start? He wrote, "Many people think that a very abstract activity, like the playing of chess, would be best."

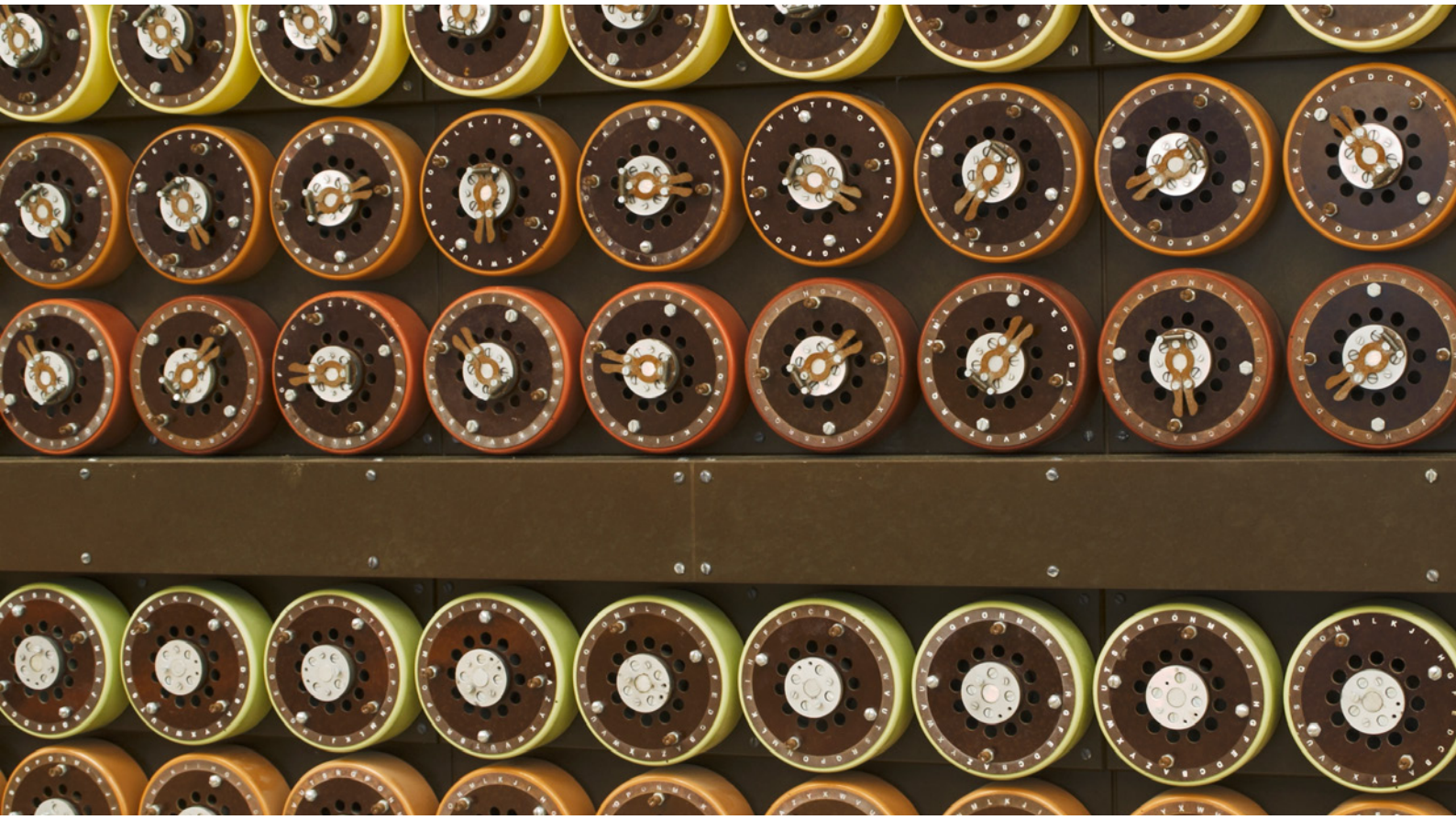
That leads to my second choice, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". We achieved superhuman performance in chess a long time ago, of course, and all the excitement transferred to developing a computer program which could play world-class Go. Once AlphaGo had done its thing, though, the Deep Mind team turned their attention back to chess and let the generic reinforcement learning algorithm developed for Go try to teach itself chess. The results are astonishing: with no provided opening book, no endgame database, no chess specific tactical knowledge or expert rules — just the basic rules of the game and a lot of self-play -— AlphaZero learned to outperform the Stockfish chess engine after four hours of training time.

Of course, chess (and Go) are constrained problems. When we take deep learning into the noisy, messy, real world, things get harder. In "Deep Learning Scaling Is Predictable, Empirically", a team from Baidu asks how to improve the state of the art in deep learning. One of the major levers that we have is to feed in more data by creating larger training data sets. But what's the relationship between model performance and size of the training data set? Will an investment in creating more data pay off in

increased accuracy? How much? Should we instead spend our effort on building better models that consume the existing data we have? The authors show that beyond a certain point, the size of a training data set and generalization error are connected by a power law. The results suggest that we should first search for model-problem fit and then scale out our data sets.

Another critical question for systems deployed in the real world is how we can have confidence they will work as expected across a wide range of inputs. In "DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars", Tian et al. consider how to test an autonomous driving system. The approaches they use could easily be applied to testing other kinds of DNNs as well. We're all familiar with the concept of code coverage, but how is your *neuron coverage* looking?

My final choice for this issue is "Deep Code Search". It's a great example of learning joint-embeddings mapping associations between two domains — in this case, natural language descriptions and snippets of code. The result is a code search engine that I'm sure InfoQ readers will relate to. It lets us ask questions such as "where in this codebase are events queued on a thread?" and be shown candidate code snippets as results.



COMPUTING MACHINERY AND INTELLIGENCE

Alan Turing's "Computing Machinery and Intelligence" is most certainly a classic paper. We've all heard of the Turing test, but have you actually read the paper in which Turing defines it?

Turing (MIND 1950)

I confess I hadn't until recently, and there's a whole lot more to it than I was expecting. Yes, it describes the Turning test (Turing at the time called it the "imitation game"), but it is also presents his thoughts on whether or not a digital computer can ever pass the test, and some remarkable observations on what it might take to build a learning machine given that Turing was writing in 1950! Plus, the writing style is a delight.

THE IMITATION GAME

I propose to consider the question, "Can machines think?" This should begin with definitions of the meaning of the terms "machine" and "think". (All quotes from Turing 1950.)

The original question, "Can machines think?" I believe to be too meaningless to deserve discussion.

- Alan Turing

To avoid the discussion degenerating into something to be settled by an opinion poll, Turing quickly refines it to a more testable proposition. The imitation game is first introduced with a man (player A) and a woman (player B) in one room, and an interrogator in another. By posing questions (in typewritten form) to the two participants, the interrogator must try to determine which of the two is the woman. The man tries to fool the interrogator into thinking he is the woman.

We now ask the question, "What will happen when a machine takes the part of A in this game?"

Can the interrogator tell the difference between human and machine? This question replaces the original "Can machines think?"

May not machines carry out something which ought to be described as thinking but which is very different from what a man does? This objection is a very strong one, but at least we can say that if, nevertheless, a machine can be constructed to play the imitation game satisfactorily, we need not be troubled by this objection.

The definition of machine is subsequently narrowed to mean a digital computer.

We are not asking whether all digital computers would do well in the game nor whether the computers at present available would do well, but whether there are imaginable computers which would do well.

After an introduction to the concept of a *universal* machine, we are permitted to focus our attention on one particular digital computer, C. We can modify it to have adequate storage, we can suitably increase its speed of action, and we can provide it with an appropriate programme. Can C be made to play satisfactorily the part of A in the imitation game, the part of B being taken by a man?

TURING'S PERSONAL BELIEF

It will simplify matters for the reader if I explain first my own beliefs in the matter. Consider first the more accurate form of the question. I believe that in about 50 years' time it will be possible to programme computers, with a storage capacity of about 10^9 (125 MB!), to make them play the imitation game so well that an average interrogator will not have more than 70% chance of making the right identification after five minutes of questioning. The original question, "Can machines think?" I believe to be too meaningless to deserve discussion.

In the original paper, Turing now proceeds to address a series of imagined objections. I'll come back to those briefly at the end, because I want to focus first on Turing's vision for learning machines.

TOWARDS LEARNING MACHINES

...[Consider] an atomic pile of less than critical size: an injected idea is to correspond to a neutron entering the pile from without. Each

such neutron will cause a certain disturbance which eventually dies away. If, however, the size of the pile is sufficiently increased, the disturbance caused by such an incoming neutron will very likely go on and on increasing until the whole pile is destroyed. Is there a corresponding phenomenon for minds, and is there one for machines?

If you present an idea to most minds, you get less than one idea back. But a smallish proportion of minds are supercritical, and an idea presented to such a mind may give rise to a whole theory with secondary, tertiary, and more remote ideas... but can a machine be made to be super-critical? It's not the storage or the speed that will be the problem argues Turing, but the programming.

Think about an adult mind: how did it arrive at its current state?

We may notice three components, (a) the initial state of the mind, say at birth, (b) the education to which it has been subjected, (c) other experience, not to be described as education, to which it has been subjected. Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one that simulates the child's!... We have thus divided our problem into two parts. The child-programme and the education process.

We won't find a good child machine at the first attempt, but Turing suggests we may be able to find one through a process of experimentation and evolution. Let the structure of the child machine be the equivalent of hereditary material, changes to the machine be mutations, and the role of natural selection be played by the judgement of the experimenter.

One may hope, however, that this process will be more expeditious than evolution. The survival of the fittest is a slow method for measuring advantages. The experimenter, by the exercise of intelligence, should be able to speed it up.

For the core of the education process, Turing suggests a form of reinforcement learning: "the machine has to be so constructed that events which shortly preceded the occurrence of a punishment-signal are unlikely to be repeated, whereas a reward-signal increased the probability of repetition of the events which led up to it." And, "these definitions do not pre-suppose any feelings on the part of the machine."

But reinforcement learning on its own will not be enough. Think how long it would take to a pupil to repeat "Casabianca" if only rewards and punishments could be given for utterances. We need some additional channels of communication — for example, a symbolic language.

Opinions may vary as to the complexity which is suitable in the child machine. One might try to make it as simple as possible consistently with the general principles. Alternatively, one might have a complete system of logical inference "built in".

It will be most important, Turing says, to regulate the order in which the rules of the logical system concerned are to be applied. For herein lies the difference between a brilliant and a footling reasoner (not the difference between a sound and a fallacious one).

Does it matter if we understand the inner workings of such a machine?

An important feature of a learning machine is that its teacher will often be very largely ignorant of quite what is going on inside, although he may still be able to some extent to predict his pupil's behaviour. This should apply most strongly to the later education of a machine arising from a child-machine of well-tried design (or programme).

It will also help if a learning machine includes a random element: "A random element is rather useful when we are searching for a solution of some problem."

OBJECTIONS TO THINKING MACHINES

Turing considers (and dismisses) nine possible objections to the idea that a machine may someday be successful in the imitation game. We don't have the space here to go into them, although they are full of character and worth reading if you have the inclination. In brief, they are:

1. The theological objection – thinking is a function of the immortal soul, and only God can create a soul. ("I am not very impressed with theological arguments. Such arguments have often been found unsatisfactory in the past." E.g., Galileo.)The "heads in the sand" objection – the consequences would be too dreadful, so let us hope and believe it's not possible. ("... Not sufficiently substantial to require refutation.")The mathematical objection – Godel's theorem and results from Church, Kleen, Rosser, and Turing show that there are limitations to the powers of discrete state machines. ("...But it has only been stated, without any sort of proof, that no such limitations apply to the human intellect. But I do not think this view can be dismissed quite so lightly.")

2. The argument from consciousness – "Yes, but can a machine really *feel*." ("This argument appears to be a denial of the validity of our test.")

3. Arguments from various disabilities – you will never be able to make a machine do X (e.g., enjoy strawberries and cream). (Mostly limited imagination based on the machines people have seen so far.)Lady Lovelace's objection – the machine can do whatever we know how to order it to perform (with an implied "only" before the "do"). A variant of this objection is that a machine can never take you by surprise. ("Machines take me by surprise with great frequency.") Argument from continuity in the nervous system – the nervous system is not a discrete-state machine ("...The interrogator will not be able to take any advantage of this difference.")

4. Argument from informality of behavior — you can't describe an explicit set of rules for all situations. ("'...There are no such rules so men cannot be machines.' The undistributed middle is glaring.")

5. Argument from extra-sensory perception — ("If telepathy is admitted... then putting the competitors into a 'telepathy-proof room' would satisfy all the requirements.")

IN CONCLUSION

We may hope that machines will eventually compete with men in all purely intellectual fields. But which are the best ones to start with? Even this is a difficult decision. Many people think that a very abstract activity, like the playing of chess, would be best. It can also be maintained that it is best to provide the machine with the best sense organs that money can buy, and then teach it to understand and speak English... Again I do not know what the right answer is, but I think both approaches should be tried.

MASTERING CHESS AND SHOGI BY SELF-PLAY WITH A GENERAL REINFORCEMENT LEARNING ALGORITHM

We looked at AlphaGo Zero last year (and the first generation of AlphaGo before that), but "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm" is still fascinating in its own right.

Silver et al. (arXiv 2017)

Recall that AlphaGo Zero learned to play Go with only knowledge of the rules and self-play. The obvious question is can you generalize that approach to other games or situations? In this paper, the DeepMind team makes another small tweak to their self-play reinforcement learning engine to give us AlphaZero, and train it to play chess, shogi, and also Go once more. It's the chess story that grabs me here — I vividly remember Deep Blue playing Gary Kasparov back in 1997, and my understanding of chess play is much better than my understanding of Go strategies and tactics. So, I can have a better appreciation for AlphaZero's play in this domain.

The game of chess represented the pinnacle of AI research over several decades. State-of-the-art programs are based on powerful engines that search many millions of positions, leveraging handcrafted domain expertise and sophisticated domain adaptations. AlphaZero is a generic reinforcement learning algorithm — originally devised for the game of Go — that achieved superior results within a few hours, searching a thousand times fewer positions, given no domain knowledge except the rules of chess. ***(All quotes from Silver et al. 2017.)***

BUT ISN'T CHESS OLD NEWS?

Computers achieved superhuman performance in chess 20 years ago, so what's the big deal about AlphaZero learning to play chess well? As with AlphaGo Zero, AlphaZero taught itself through self-play with only knowledge of the rules — no opening book, no endgame database, no chess-specific tactical knowledge, etc. And while trans-

> ***It took just four hours for AlphaZero to outperform Stockfish, and eight hours to beat AlphaGo Lee.***

lation-invariant nature of Go has a natural match with the structure of convolutional networks, the fit with chess and shogi is not so clear cut:

Chess and shogi are, arguably, less innately suited to AlphaGo's neural network architectures. The rules are position-dependent (e.g., pawns may move two steps forward from the second rank and promote on the eighth rank) and asymmetric (e.g., pawns only move forward and castling is different on kingside and queenside). The rules include long-range interactions (e.g., the queen may traverse the board in one move or checkmate the king from the far side of the board.

ALPHAZERO PLAYING CHESS

Trained AlphaZero engines played chess against Stockfish, shogi against Elmo, and Go against the previous version of AlphaGo Zero. In each case, 100 game matches were played with a time limit of one minute per move. AlphaZero used four TPUs, as did AlphaGo Zero; Stockfish and Elmo played at their strongest levels using 64 threads and a 1-GB hash size.

AlphaZero convincingly defeated all opponents, losing zero games to Stockfish and eight games to Elmo, as well as defeating the previous version of AlphaGo Zero.

Game	White	Black	Win	Draw	Loss
Chess	*AlphaZero*	*Stockfish*	25	25	0
	Stockfish	*AlphaZero*	3	47	0
Shogi	*AlphaZero*	*Elmo*	43	2	5
	Elmo	*AlphaZero*	47	0	3
Go	*AlphaZero*	*AG0 3-day*	31	–	19
	AG0 3-day	*AlphaZero*	29	–	21

Table 1: Tournament evaluation of *AlphaZero* in chess, shogi, and Go, as games won, drawn or lost from *AlphaZero*'s perspective, in 100 game matches against *Stockfish*, *Elmo*, and the previously published *AlphaGo Zero* after 3 days of training. Each program was given 1 minute of thinking time per move.

It took just four hours for AlphaZero to outperform Stockfish, and eight hours to beat AlphaGo Lee. That's elapsed game time, of course. The researchers used 5,000 TPUs to generate self-play games and 64 TPUs to train the networks. So perhaps a more accurate statement would be "it took approximately 20,000 TPU-hours" for AlphaZero to outperform Stockfish.

The headlines are impressive. But it's the way that AlphaZero plays chess that really captures my attention. AlphaZero seems to play in a much more human style. It has élan, it has flair. To see what I mean, it's most instructive to look at some moments from the games.

Here's a YouTube video of IM Daniel Rensch's commentary on some of the match highlights

I also really enjoyed agadmator's video analysis of this game where AlphaZero trades a pawn to leave black's bishop in a weak position, crippling black for much of the game.

Bearing in mind that AlphaZero has no openings database, it's also interesting to see that it rediscovers popular openings all by itself, seeming to end up with a preference for the English Opening and the Queen's Gambit. (see next page)

Each of these openings is independently discovered and played frequently by AlphaZero during self-play training. When starting from each human opening, AlphaZero convincingly defeated Stockfish, suggesting it has indeed mastered a wide spectrum of chess play.

There's some evidence to back my assertion that AlphaZero's play feels more like human play, too: AlphaZero uses a Monte Carlo tree-search (MCTS) algorithm that searches 80,000 positions per second in chess. Granted, that's many more positions than any human — but compared to the 70 million positions per second that Stockfish crunches through, it's tiny. Three orders of magnitude fewer! So AlphaZero must genuinely have a (if you'll excuse the pun) deeper understanding of the chess positions in order to focus that much more limited energy in the places where it matters the most:

AlphaZero compensates for the lower number of evaluations by using its deep neural network to focus more selectively on the most promising variations — arguably a more 'human-like' approach to search...

UNDER THE HOOD: FROM ALPHAGO ZERO TO ALPHAZERO

So how does AlphaZero do it? The heart of AlphaZero is very similar to AlphaGo Zero, which I looked at last year, but in an even more general form. There is the same single-network structure that outputs both a vector of move probabilities and a scalar estimate of the expected outcome from the current position. And the network is similarly trained from self-play.

Games are played by selecting moves for both players using Monte Carlo tree search.... At the end of the game, the terminal position is scored according to the rules of the game to compute the game outcome.... The neural network parameters θ are updated so as to minimise the error between the predicted outcome and the game outcome, and to maximise the similarity of the policy vector to the search probabilities.

The main differences to AlphaGo Zero are as follows:

- AlphaGo Zero assumes binary win/loss outcomes; AlphaZero also takes draws into account.
- The AlphaGo family exploits Go's invariance to rotation and reflection, generating eight symmetries for each position and randomly selecting rotations or reflections before training. AlphaZero cannot do this as chess (and shogi) do not have the same invariance properties.
- AlphaZero continually updates a single neural network during self-play. In contrast, AlphaGo Zero moved forward using a batch process whereby the performance of a new player after a training iteration was measured against the previous best and only took over as the new generation source for self-play games if it won by a margin of 55%.
- AlphaGo Zero tuned its hyperparameters using Bayesian optimisation. AlphaZero simply reuses the same hyperparameters for all games without game-specific tuning.

And, of course, AlphaZero needs a board-input representation suitable for the games it is playing. For chess, the board is represented by 16 eight-by-eight planes. Six eight-by-eight planes represent the position of the white pieces (one for the pawn positions, one for knights, one for bishops, one for rooks, one for the king, and one for the queen) and a similar set to represent the positions of the black pieces. In addition, there are a number of constant-value inputs for the

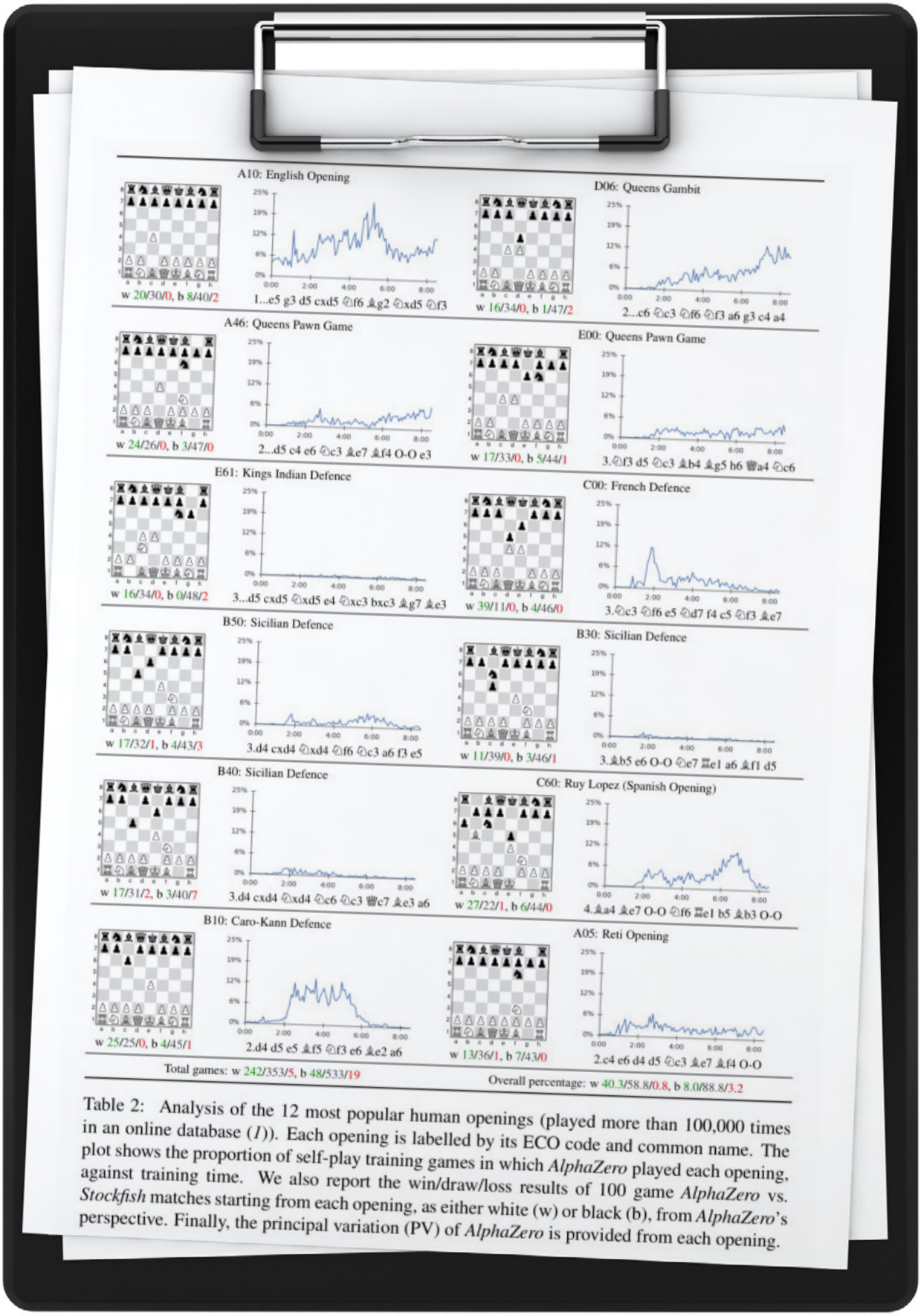


Table 2: Analysis of the 12 most popular human openings (played more than 100,000 times in an online database (*1*)). Each opening is labelled by its ECO code and common name. The plot shows the proportion of self-play training games in which *AlphaZero* played each opening, against training time. We also report the win/draw/loss results of 100 game *AlphaZero* vs. *Stockfish* matches starting from each opening, as either white (w) or black (b), from *AlphaZero*'s perspective. Finally, the principal variation (PV) of *AlphaZero* is provided from each opening.

player's color, total move count, legality of castling, move repetition count (three repetitions is a draw in chess), and the number of moves without progress (50 moves without progress is a draw).

Go		Chess		Shogi	
Feature	Planes	Feature	Planes	Feature	Planes
P1 stone	1	P1 piece	6	P1 piece	14
P2 stone	1	P2 piece	6	P2 piece	14
		Repetitions	2	Repetitions	3
				P1 prisoner count	7
				P2 prisoner count	7
Colour	1	Colour	1	Colour	1
		Total move count	1	Total move count	1
		P1 castling	2		
		P2 castling	2		
		No-progress count	1		
Total	17	Total	119	Total	362

Table S1: Input features used by *AlphaZero* in Go, Chess and Shogi respectively. The first set of features are repeated for each position in a $T = 8$-step history. Counts are represented by a single real-valued input; other input features are represented by a one-hot encoding using the specified number of binary input planes. The current player is denoted by P1 and the opponent by P2.

The policy encodes a probability distribution over 4,672 possible moves in an 8x8x73 stack of planes.

Each of the 8x8 positions identifies the square from which to "pick up" a piece. The first 56 planes encode possible "queen moves" for any piece: a number of squares [1..7] in which the piece will be moved, along one of the eight relative compass directions {N, NE, E, SE, S, SW, W, NW}. The next eight planes encode possible knight moves for that piece. The final nine planes encode possible underpromotions for pawn moves or captures in two possible diagonals, to knight, bishop, or rook respectively. Other pawn moves or captures from the seventh rank are promoted to a queen.

Bearing in mind that AlphaZero has no openings database, it's also interesting to see that it rediscovers popular openings all by itself, seeming to end up with a preference for the English Opening and the Queen's Gambit.

DEEP-LEARNING SCALING IS PREDICTABLE, EMPIRICALLY

"Deep learning scaling is predictable, empirically" is a wonderful study with far-reaching implications that could even impact company strategies in some cases. (With thanks to Nathan Benaich for highlighting this paper in his excellent summary of the AI world in 1Q18)

Hestness et al. (arXiv 2017)

It starts with a simple question of how we can improve the state of the art in deep learning (DL). We have three main lines of attack:

1. We can search for improved model architectures.
2. We can scale computation.
3. We can create larger training data sets.

As DL application domains grow, we would like a deeper understanding of the relationships between training set size, computational scale, and model accuracy improvements to advance the state of the art. (All quotes from Hestness et al. 2017.)

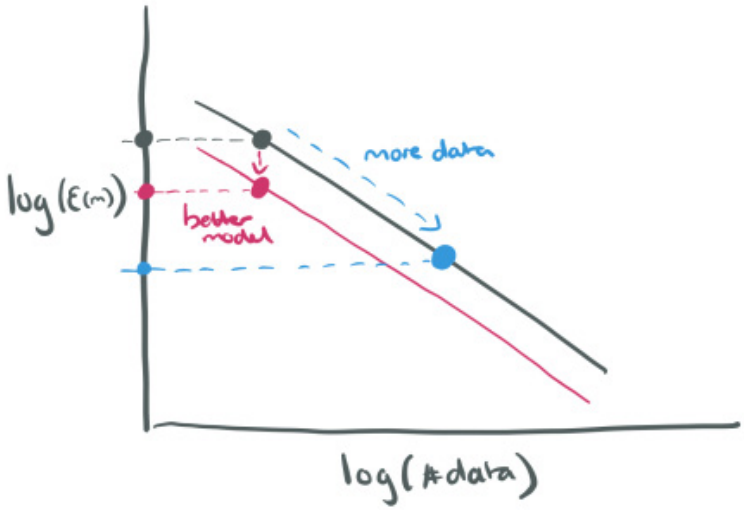
Finding better model architectures often depends on unreliable epiphany and, as the results show, has limited impact compared to increasing the amount of data available. We've known this for some time of course, including from the 2009 Google paper, "The Unreasonable Effectiveness of Data". The results from Hestness et al. help us to quantify the data advantage across a range of deep-learning applications. The key to understanding is captured in the following equation:

$$\epsilon(m) \sim \alpha m^{\beta_g} + \gamma$$

The equation says that the generalization error ϵ, as a function

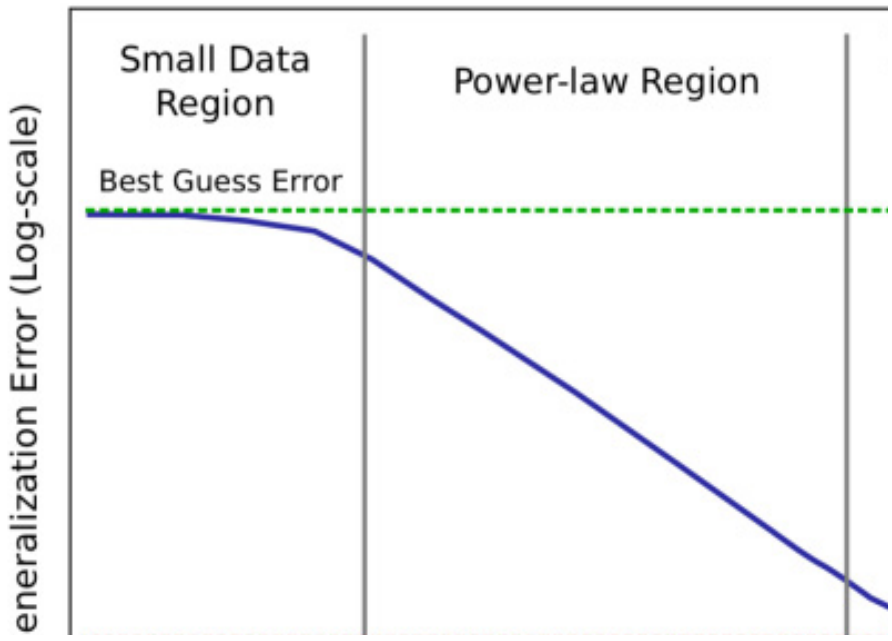
of the amount of training data m, follows a power law with exponent β_g. Empirically, β_g usually seems to fall inside the range of -0.07 to -0.35. With error of course, lower is better, hence the negative exponents. We normally plot power laws on log-log graphs, where they result in straight lines with the gradient indicating the exponent.

Making better models can move the y-intercept down (until we reach the irreducible error level) but doesn't seem to impact the power-law coefficient. On the other hand, more data puts us on a power-law path of improvement.



THE THREE LEARNING ZONES

The learning curves for real applications can be broken down into three regions:



- The **small-data region** is where models struggle to learn from insufficient data and models can only perform as well as best or random guessing.

- The middle region is the **power-law region**, where the power-law exponent defines the steepness of the curve (slope on a log-log scale). The exponent is an indicator of the difficulty for models to represent the data generating function. ***Results in this paper indicate that the power-law exponent is unlikely to be easily predicted with prior theory and probably dependent on aspects of the problem domain or data distribution.***

- The **irreducible-error region** is the non-zero lower-bound error past which models will be unable to improve. With sufficiently large training sets, models saturate in this region.

ON MODEL SIZE

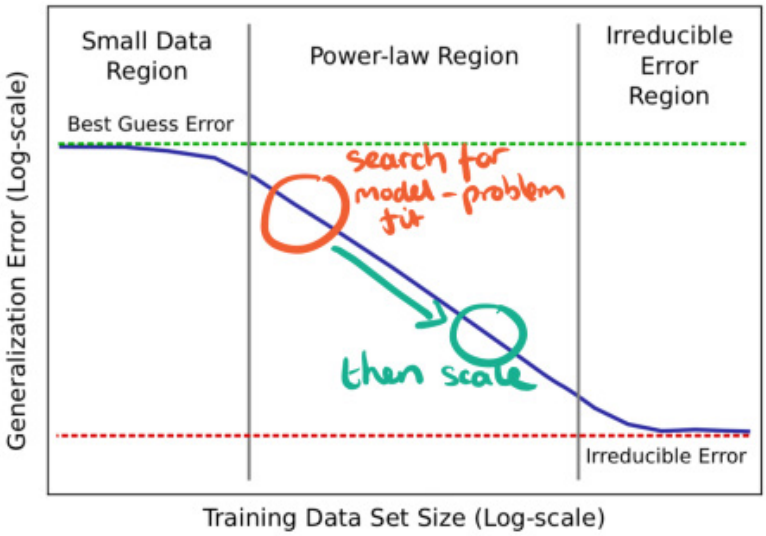
We expect the number of model parameters to fit a data set should follow $s(m) \sim \alpha m^{\beta p}$ where $s(m)$ is the required model size to fit a training set of size m, and $\beta p \in [0.5,1]$.

Best-fit models grow sub-linearly in training shard size. Higher values of β_p indicate models that make less effective use of extra parameters on larger data sets. Despite model size scaling differences though, "for a given model architecture, we can accurately predict the model size that will best fit increasingly larger data sets."

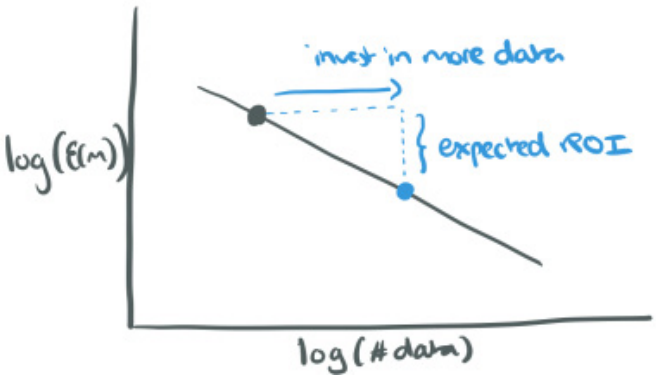
IMPLICATIONS OF THE POWER LAW

Predictable learning curves and model size scaling indicate some significant implications on how DL could proceed. For machine-learning practitioners and researchers, predictable scaling can aid model and optimization debugging and iteration time, and offer a way to estimate the most impactful next steps to improve model accuracy. Operationally, predictable curves can guide decision making about whether or how to grow data sets or computation.

One interesting consequence is that model exploration can be done on smaller data sets (and hence faster/cheaper). The data set needs to be large enough to show accuracy in the power-law region of the curve. Then the most promising models can be scaled to larger data sets to ensure proportional accuracy gains. This works because growing training sets and models is likely to result in the same relative gains across models. When building a company, we look for product-market fit before scaling the business. In deep learning, it seems the analogy is to look for model-problem fit before scaling, a "search then scale" strategy:



If you need to make a business decision about the return on investment, or likely accuracy improvement, from investing in the collection of more data, the power law can help you predict returns:



Conversely, if generalisation error within the power-law region drifts from the power-law predictions, it's a clue that increasing model size or a more extensive hyperparameter search (i.e., more compute) might help:

...Predictable learning and model size curves may offer a way to project the compute requirements to reach a particular accuracy level.

CAN YOU BEAT THE POWER LAW?

Model architecture improvements (e.g., increasing model depth) seem only to shift learning curves down, but not improve the power-law exponent.

We have yet to find factors that affect the power-law exponent. To beat the power law as we increase data set size, models would need to learn more concepts with successively less data. In other words, models must successively extract more marginal information from each additional training sample.

If we can find ways to improve the power-law exponent though, then the potential accuracy improvements in some problem domains are immense.

THE EMPIRICAL DATA

The empirical data to back all this up was collected by testing various training-data sizes (in powers of two) with state-of-the-art deep-learning models in a number of different domains.

Here are the learning curves for neural machine translation. On the right, we can see results for the best-fit model at each size of training set. As training-set size grows, the empirical error tends away from the power-law trend, and a more exhaustive hyperparameter search would be needed to bring it back in line.

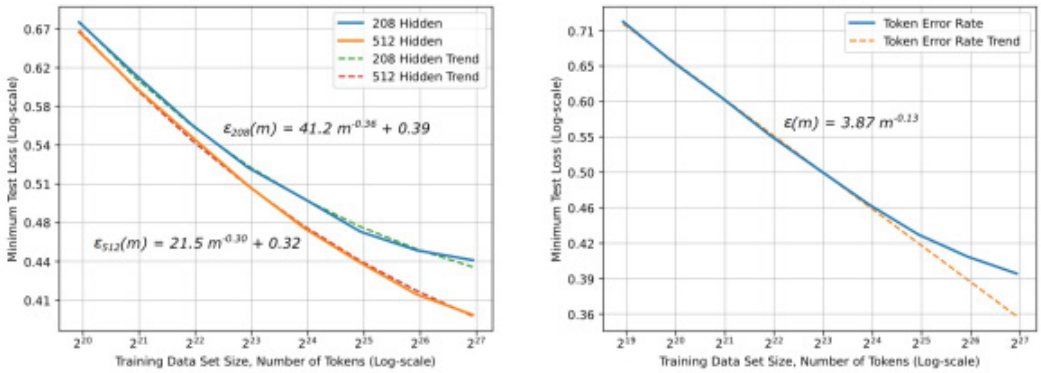


Figure 1: Neural machine translation learning curves. Left: the learning curves for separate models follow $\varepsilon(m) = \alpha m^{\beta_g} + \gamma$. Right: composite learning curve of best-fit model at each data set size.

The next domain is word language models. A variety of model architectures are used, and although they differ appreciably, they all show the same learning-curve profile as characterized by the power-law exponent.

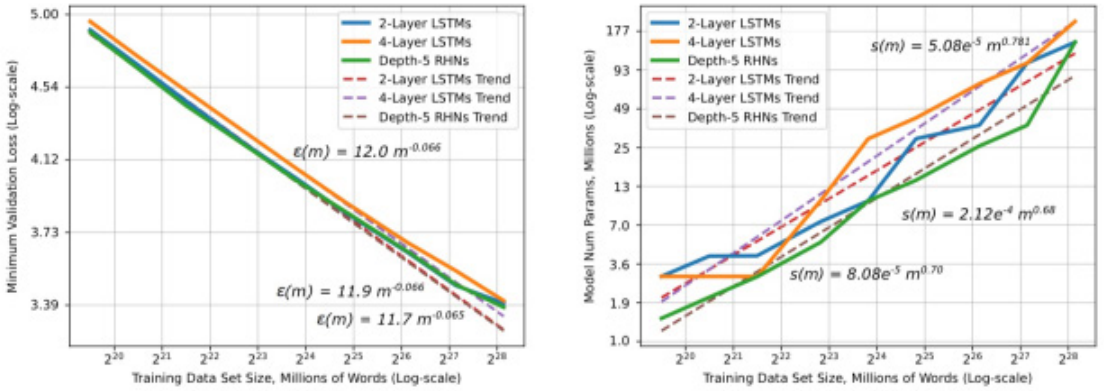


Figure 2: Learning curve and model size results and trends for word language models.

And here are the results for character language models:

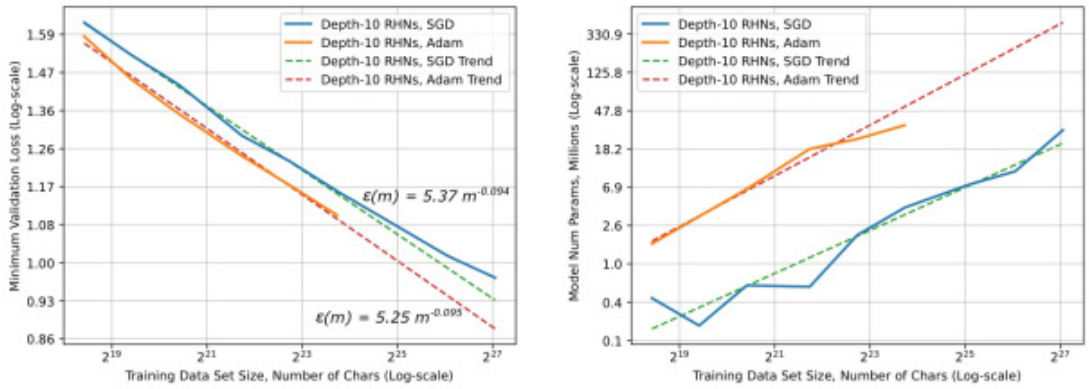


Figure 3: Learning curve and model size results and trends for character language models.

With image classification, we see the small-data region appear on the plots when there is insufficient data to learn a good classifier:

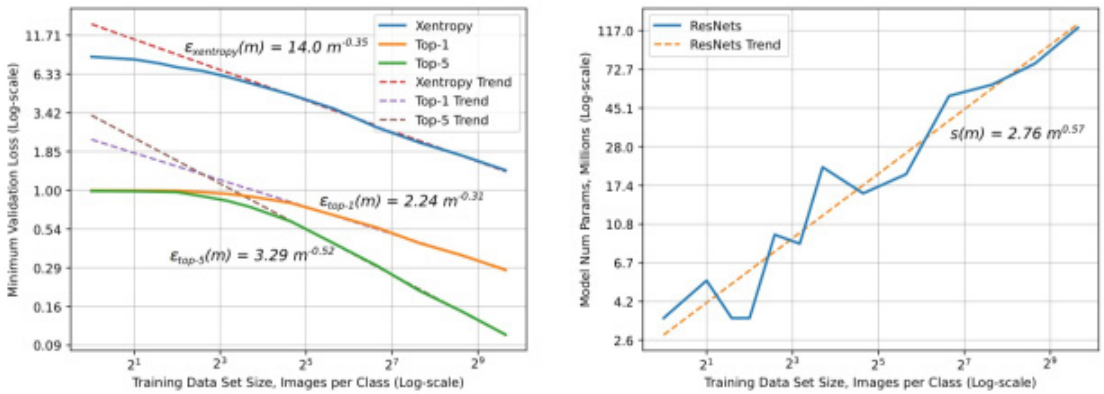


Figure 4: Learning curve and model size results and trends for ResNet image classification.

Finally, here's speech recognition:

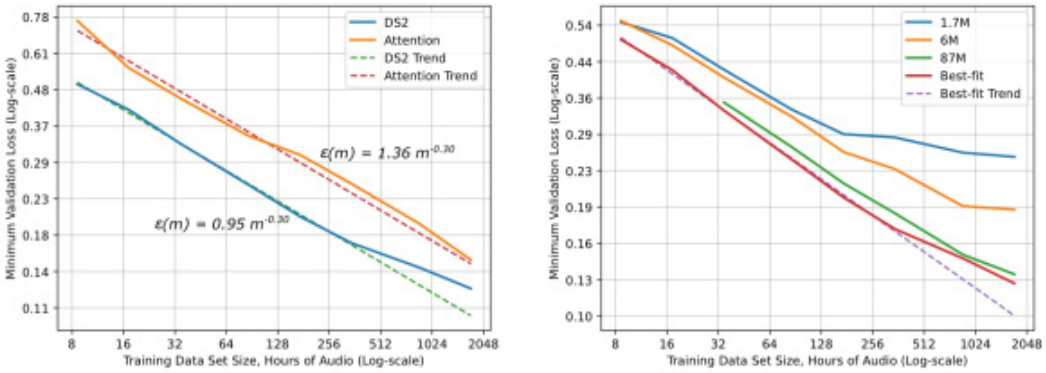


Figure 5: Learning curves for DS2 and attention speech models (left), and learning curves for various DS2 model sizes, 1.7M to 87M parameters (right).

We empirically validate that DL model accuracy improves as a power law as we grow training sets for state-of-the-art (SOTA) model architectures in four machine-learning domains: machine translation, language modeling, image processing, and speech recognition. These power-law learning curves exist across all tested domains, model architectures, optimizers, and loss functions.

DEEP CODE SEARCH

"Deep Code Search" takes on the problem with searching for code, which is that the query (e.g., "read an object from xml") doesn't look very much like the source-code snippets that are the intended results...

Gu et al. (ICSE 2018)

...such as:

```
public static < S > S deserialize(Class c, File xml) {
    try {
        JAXBContext context = JAXBContext.newInstance(c);
        Unmarshaller unmarshaller = context.createUnmarshaller();
        S deserialized = (S) unmarshaller.unmarshal(xml);
        return deserialized;
    } catch (JAXBException ex) {
        log.error("Error-deserializing-object-from-XML", ex);
        return null;
    }
}
```

Compared to Lucene-powered code-search tools and the recently proposed state- of-the-art CodeHow search tool, CODEnn gives excellent results.

That's why we have Stack Overflow! Stack Overflow can help with "how to"-style queries, but it can't help with searches inside codebases you care about — e.g., "where in this codebase are events queued on a thread?"

...an effective code search engine should be able to understand the semantic meanings of natural language queries and source code in order to improve the accuracy of code search. (All quotes from Gu et al. 2018.)

DeepCS is just such a search engine for code, based on the CODEnn (Code-description-embedding neural network) model. During training, it takes code snippets (methods) and corresponding natural-language descriptions (from the method comments) and learns a joint embedding — i.e., it learns embeddings such that a method description and its corresponding code snippet are both mapped to a similar point in the same shared embedding space. Given a natural-language query, it can then embed the query in vector space and look for nearby code snippets. Compared to Lucene-powered code-search tools and the recently proposed state-of-the-art CodeHow search tool, CODEnn gives excellent results.

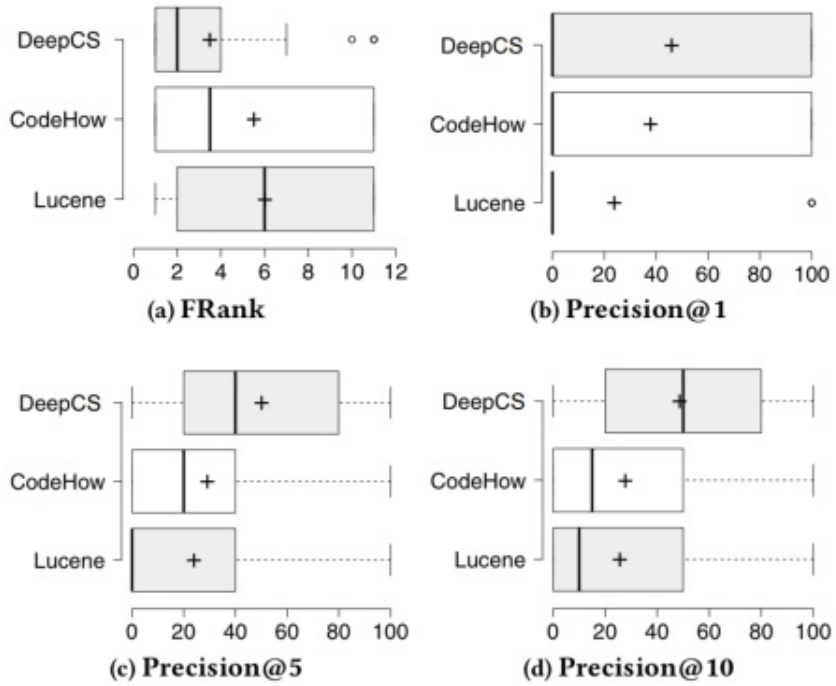
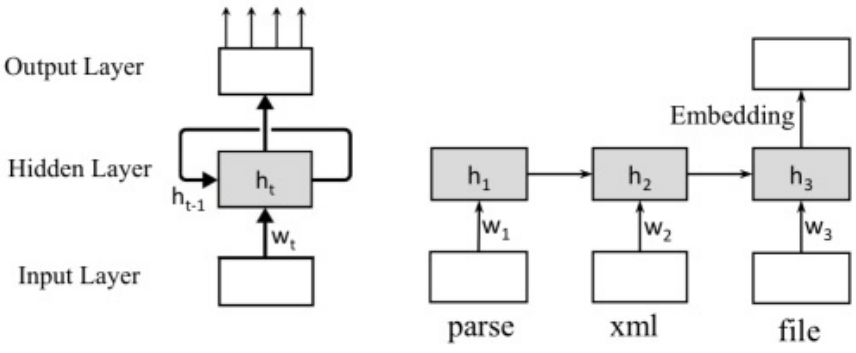


Figure 8: The statistical comparison of FRank and Precison@k for three code search approaches

EMBEDDINGS

One of the most popular posts on The Morning Paper blog has been "The amazing power of word vectors", which described the process of turning words into vectors. To turn a sequence of words into a vector, one common approach is to use a recurrent neural network (RNN) (e.g., long short-term memory — LSTM). For example, given a sentence such as "parse xml file", the RNN reads the first word, "parse", maps it into a vector w_1, and then computes the RNN hidden state h_1 using w_1. Then it reads the second word, "xml", maps that into word vector w_2 and updates the hidden state h_1 to h_2. It keeps going in this manner until it reaches the end of the sentence and then uses the final hidden state (h_3 in this example) as the sentence embedding.



(a) RNN Structure (b) RNN for sentence embedding

Figure 1: Illustration of the RNN Sentence Embedding

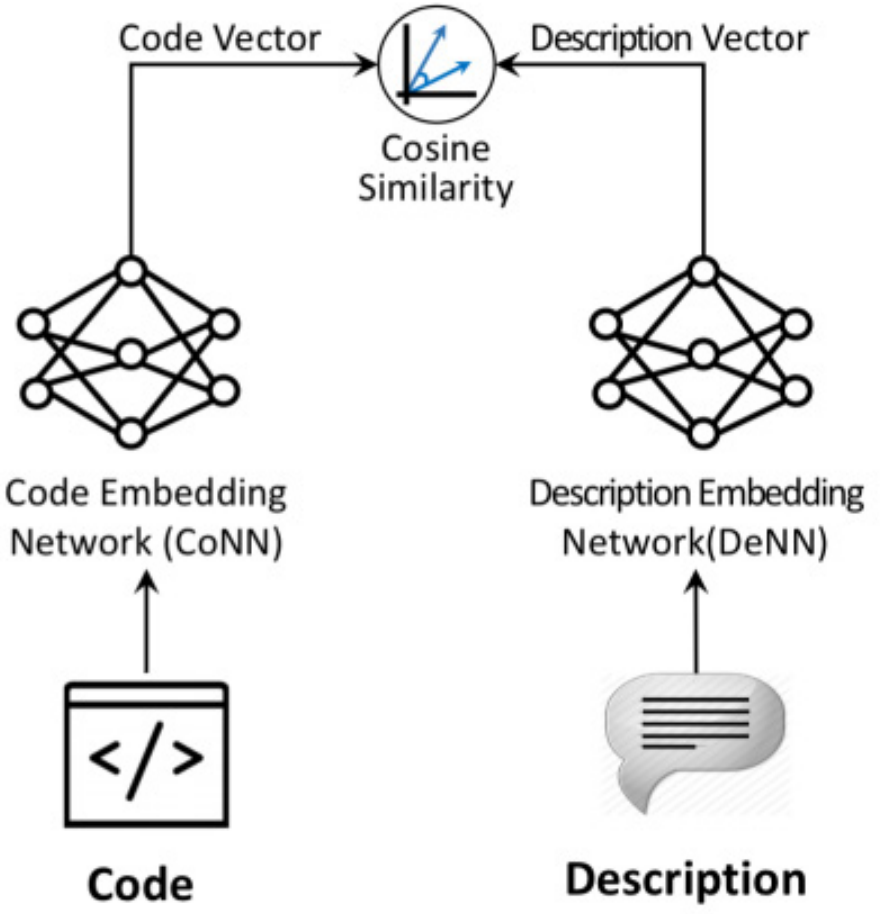
We don't want just any old embedding though. We want to learn embeddings such that code snippets (think of them like code sentences for now; we'll get to the details shortly) and their corresponding descriptions have nearby embeddings.

Joint Embedding, also known as multi-model embedding, is a technique to jointly embed/correlate heterogeneous data into a unified vector space so that semantically similar concepts across the two modalities occupy nearby regions of the space.

When we're training, we use a similarity measure (e.g., cosine) in the loss function to encourage the joint mapping.

CODENN HIGH-LEVEL ARCHITECTURE

CODEnn has three main components: a code embedding network to embed source-code snippets in vectors; a description-embedding network to embed natural-language descriptions into vectors; and a similarity module (cosine) to measure the degree of similarity between code and descriptions. At a high level, it looks like the diagram to the right.



If we zoom in one level, we can start to see the details of the network constructions, as shown at the top of the next page.

THE CODE-EMBEDDING NETWORK

A code snippet (a method) is turned into a tuple (M, A, T) where M is a sequence of camel-case split tokens in the method name, A is an API sequence (the method invocations made by the method body), and T is the set of tokens in the snippet.

The method name and API invocation sequences are both embedded (separately) into vectors using RNNs with max pooling. The tokens have no strict order, so they are embedded using a conventional multilayer perceptron. The three resulting vectors are then fused into one vector through a fully connected layer.

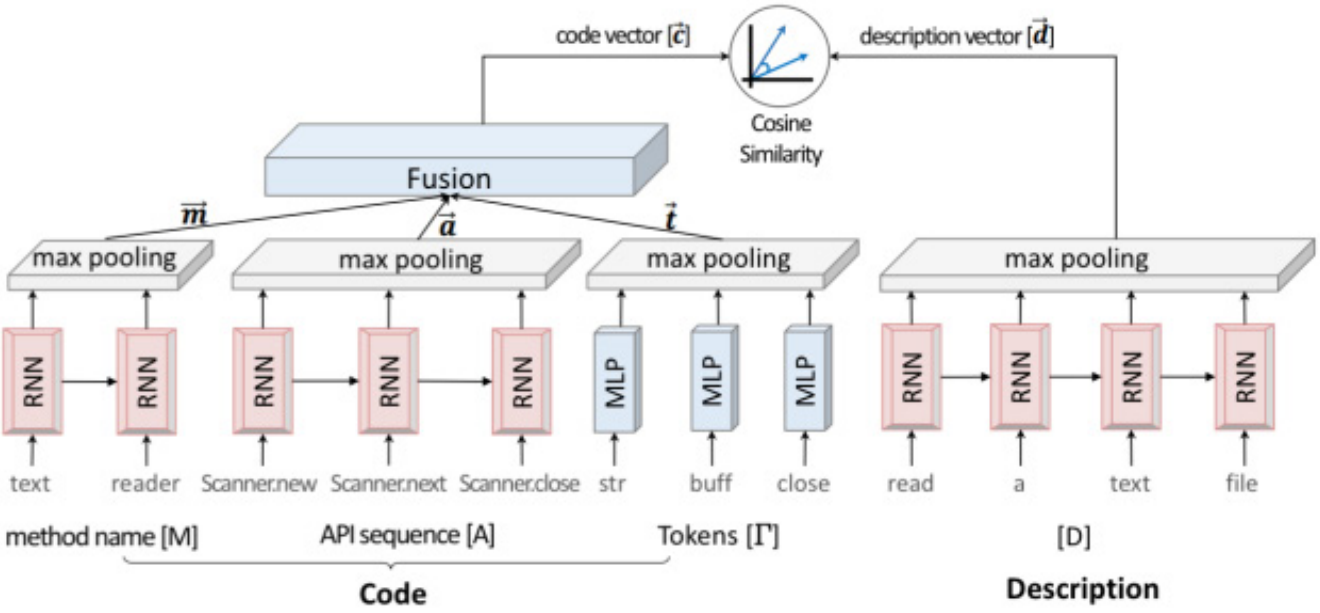
THE DESCRIPTION-EMBEDDING NETWORK

The description embedding network uses a RNN with max pooling to encode the natural-language description from the (first sentence of) the method comment.

TRAINING

During training, training instances are given as triples (C, D+, D-), where C is a code snippet, D+ is the correct (actual) description of C, and D- is an incorrect description chosen randomly from the pool of all descriptions. The loss function seeks to maximize the cosine similarity between C and D+, and make the distance between C and D- as large as possible. (see Figure 5, on next page)

The training corpus is based on open-source Java projects on



GitHub, resulting in a body of just over 18 million commented Java methods.

For each Java method, we use the method declaration as the code element and the first sentence of its documentation comment as its natural-language description. According to the Javadoc guidance, the first sentence is usually a summary of a method.

The method bodies are parsed using the Eclipse JDT compiler. (Note, some of the tools from source{d} may also be useful here).

All the RNNs are embodied as LSTMs with 200 hidden units in each direction, and word embeddings have 100 dimensions.

INDEXING AND QUERYING

To index a codebase, DeepCS embeds all code snippets in the codebase into vectors using offline processing. Then, during online searching, DeepCS embeds the natural-language user query, and estimates the cosine similarities between the query embedding and precomputed code-snippet embeddings. The top K closest code snippets are returned as the query results.

EVALUATION

Evaluation is done using a search codebase constructed from ~10 thousand Java projects on GitHub (different to the training set), with at least 20 stars each. All code is indexed (including methods without any comments), resulting in just over 16 million indexed methods.

We build a benchmark of queries from the top 50 voted Java programming questions in Stack Overflow. To achieve so, we browse the list of Java-tagged questions in Stack Overflow and sort them according to the votes that each one receives.

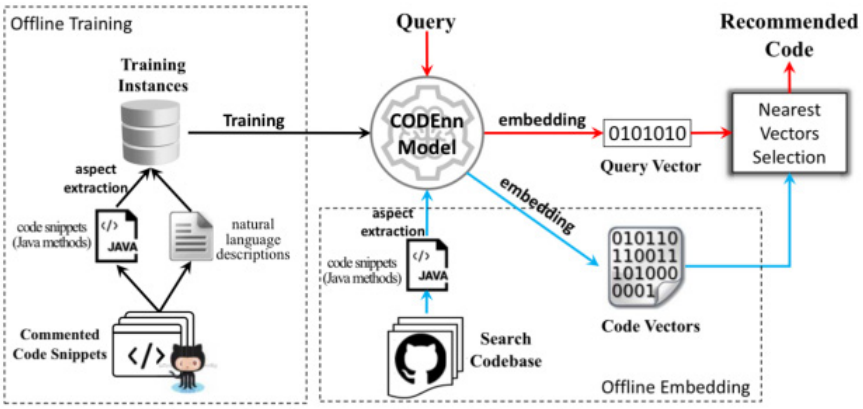


Figure 5: The overall workflow of DeepCS

To qualify, the questions must be about a concrete Java programming task and include a code snippet in the accepted answer. The 50 such resulting questions are shown in the following table. In the right-hand column, we can see the FRank (rank of the first hit result in the result list) for DeepCS, CodeHow, and a conventional Lucene-based search tool.

Table 1: Benchmark Queries and Evaluation Results (NF: Not Found within the top 10 returned results LC:Lucene CH:CodeHow DCS:DeepCS)

No.	Question ID	Query	FRank		
			LC	CH	DCS
1	309424	convert an inputstream to a string	2	1	1
2	157944	create arraylist from array	NF	NF	2
3	1066589	iterate through a hashmap	NF	4	1
4	363681	generating random integers in a specific range	NF	6	2
5	5585779	converting string to int in java	NF	10	1
6	1005073	initialization of an array in one line	NF	4	1
7	1128723	how can I test if an array contains a certain value	6	6	1
8	604424	lookup enum by string value	1	NF	10
9	886955	breaking out of nested loops in java	NF	NF	NF
10	1200621	how to declare an array	NF	NF	4
11	41107	how to generate a random alpha-numeric string	NF	1	1
12	409784	what is the simplest way to print a java array	6	NF	1
13	109383	sort a map by values	NF	1	3
14	295579	fastest way to determine if an integer's square root is an integer	NF	NF	NF
15	80476	how can I concatenate two arrays in java	NF	1	1
16	326369	how do I create a java string from the contents of a file	8	NF	5
17	1149703	how can I convert a stack trace to a string	3	1	2
18	513832	how do I compare strings in java	1	3	1
19	3481828	how to split a string in java	1	1	1
20	2885173	how to create a file and write to a file in java	2	1	NF
21	507602	how can I initialise a static map	7	1	2
22	223918	iterating through a collection, avoiding concurrentmodificationexception when removing in loop	3	3	2
23	415953	how can I generate an md5 hash	1	3	6
24	1069066	get current stack trace in java	3	1	1
25	2784514	sort arraylist of custom objects by property	1	1	1
26	153724	how to round a number to n decimal places in java	1	1	4
27	473282	how can I pad an integers with zeros on the left	NF	3	1
28	529085	how to create a generic array in java	NF	NF	3
29	4716503	reading a plain text file in java	4	NF	7
30	1104975	a for loop to iterate over enum in java	NF	NF	NF
31	3076078	check if at least two out of three booleans are true	NF	NF	NF
32	4105331	how do I convert from int to string	2	1	NF
33	8172420	how to convert a char to a string in java	5	10	3
34	1816673	how do I check if a file exists in java	1	2	1
35	4216745	java string to date conversion	6	NF	1
36	1264709	convert inputstream to byte array in java	7	5	1
37	1102891	how to check if a string is numeric in java	1	NF	2
38	869033	how do I copy an object in java	2	1	1
39	180158	how do I time a method's execution in java	NF	NF	2
40	5868369	how to read a large text file line by line using java	1	1	1
41	858572	how to make a new list in java	2	1	1
42	1625234	how to append text to an existing file in java	3	1	1
43	2201925	converting iso 8601-compliant string to date	3	1	1
44	122105	what is the best way to filter a java collection	NF	9	2
45	5455794	removing whitespace from strings in java	NF	3	1
46	225337	how do I split a string with any whitespace chars as delimiters	1	1	2
47	52353	in java, what is the best way to determine the size of an object	NF	NF	NF
48	160970	how do I invoke a java method when given the method name as a string	3	1	2
49	207947	how do I get a platform dependent new line character	1	NF	10
50	1026723	how to convert a map to list in java	6	NF	1

Here are some representative query results.

"Queue an event to be run on the thread" versus "run an event on a thread queue" (both have similar words in the query, but are quite different):

```
public boolean enqueue(EventHandler handler, Event event) {
    synchronized(monitor) {
        ......
        handlers[tail] = handler;
        events[tail] = event;
        tail++;
        if (handlers.length <= tail)
            tail = 0;
        monitor.notify();
    }
    return true;
}
```

(a) The third result of the query "queue an event to be run on the thread"

```
public void run() {
    while (!stop) {
        DynamicModelEvent evt;
        while ((evt = eventQueue.poll()) != null) {
            for (DynamicModelListener l: listeners.toArray(
                                        new DynamicModelListener[0]))
                l.dynamicModelChanged(evt);
        }
        ......
    }
}
```

(b) The first result of the query "run an event on the thread queue"

Figure 9: Examples showing the query understanding

"Get the content of an input stream as a string using a specified character encoding":

```
public static String toStringWithEncoding(
                InputStream inputStream, String encoding) {
    if (inputStream == null)
        throw new IllegalArgumentException(
                    "inputStream-should-not-be-null");
    char[] buffer = new char[BUFFER_SIZE];
    StringBuffer stringBuffer = new StringBuffer();
    BufferedReader bufferedReader = new BufferedReader(
            new InputStreamReader(inputStream, encoding), BUFFER_SIZE);
    int character = -1;
    ......
    return stringBuffer.toString();
}
```

Figure 10: An example showing the search robustness – The first result of the query "get the content of an input stream as a string using a specified character encoding"

"Read an object from an xml file" (note that the words "xml", "object", and "read" don't appear in the result, but DeepCS still finds it):

```
public static < S > S deserialize(Class c, File xml) {
    try {
        JAXBContext context = JAXBContext.newInstance(c);
        Unmarshaller unmarshaller = context.createUnmarshaller();
        S deserialized = (S) unmarshaller.unmarshal(xml);
        return deserialized;
    } catch (JAXBException ex) {
        log.error("Error-deserializing-object-from-XML", ex);
        return null;
    }
}
```

(a) The first result of the query "read an object from an xml file"

"Play a song" (another example of associative search, with which DeepCS can recommend results with semantically related words such as "audio"):

```
public void playVoice(int clearedLines) throws Exception {
    int audiosAvailable = audioLibrary.get(clearedLines).size();
    int audioIndex = rand.nextInt(audiosAvailable);
    audioLibrary.get(clearedLines).get(audioIndex).play();
}
```

(b) The second result of the query "play a song"

DeepCS isn't perfect of course, and sometimes ranks partially relevant results higher than exact matching ones.

This is because DeepCS ranks results by just considering their semantic vectors. In future work, more code features (such as programming context) could be considered in our model to further adjust the results.

DEEPTEST: AUTOMATED TESTING OF DNN-DRIVEN AUTONOMOUS CARS

Here's a look at "DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars".

Tian et al. (ICSE 2018)

How do we test a deep neural network (DNN)? We've seen plenty of examples of adversarial attacks in previous editions of The Morning Paper, but we couldn't really say that generating adversarial images is enough to give us confidence in the overall behavior of a model under all operating conditions. Adversarial images approach things from a "think like an attacker" mindset. But we want to think like a tester — for example, DeepXplore uses model ensembles to find differences in outputs that suggest bugs.

The importance of testing DNNs is especially obvious when it comes to applications such as autonomous driving. DeepTest uses several of the ideas from DeepXplore and looks specifically at testing autonomous-driving systems. I think you could apply the DeepTest techniques to test other kinds of DNNs as well.

...despite the tremendous progress, just like traditional software, DNN-based software, including the ones used for autonomous driving, often demonstrate incorrect/unexpected

corner-case behaviors that lead to dangerous consequences like a fatal collision. ***(All quotes from Tian et al. 2018.)***

DeepTest is a system designed to aid in the testing of autonomous-driving models. When used to test three of top performing DNNs from the Udacity self-driving car challenge, it revealed thousands of erroneous behaviors, many of which could lead to potentially fatal crashes.

TESTING CHALLENGES IN AUTONOMOUS DRIVING

We're interested in testing the DNNs at the core of autonomous-driving systems. These take inputs from a variety of sensors and actuate car systems such as steering, braking, and accelerating.

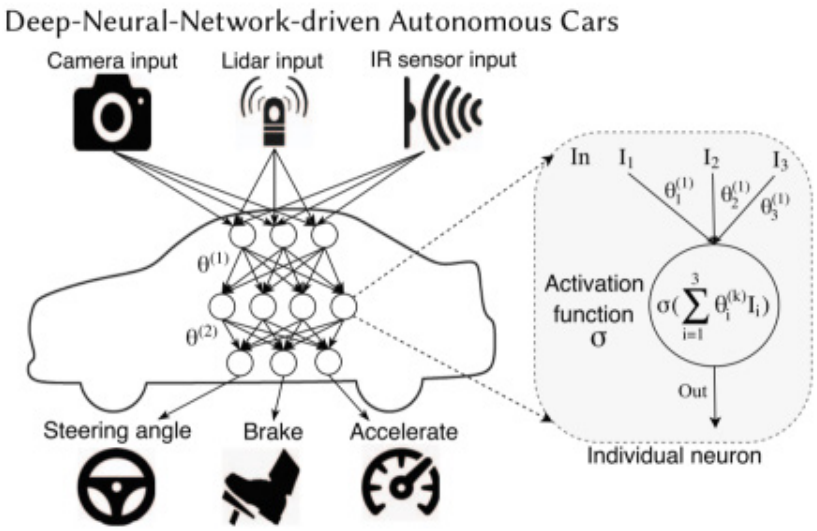


Figure 2: A simple autonomous car DNN that takes inputs from camera, light detection and ranging sensor (LiDAR), and IR (infrared) sensor, and outputs steering angle, braking decision, and acceleration decision. The DNN shown here essentially models the function $\sigma(\theta^{(2)} \cdot \sigma(\theta^{(1)} \cdot x))$ where θs represent the weights of the edges and σ is the activation function. The details of the computations performed inside a single neuron are shown on the right.

This paper focuses on the camera input and the steering-angle output. We can't apply traditional measures such as statement coverage to understand how well tested a DNN is, so we need to find an alternate metric. Borrowing from DeepXplore, DeepTest uses a notion of neuron coverage. Another interesting challenge is how we know whether the output of the model is correct in any given scenario. DeepXplore introduced the notion of using an ensemble of models to detect models that make unusual predictions for a given input. DeepTest has a neat twist on this, using an ensemble of inputs which should all lead to the same output (e.g., the same road in different weather and visibility conditions) to detect erroneous outputs.

MEASURING TEST COVERAGE

The input-output space (i.e., all possible combinations of inputs and outputs) of a complex system like an autonomous vehicle is too large for exhaustive exploration. Therefore, we must devise a systematic way of partitioning the space into different equivalence classes by picking one sample from each of them. In this paper, we leverage neuron coverage as a mechanism for partitioning the input space based on the assumption that all inputs that have similar neuron coverage are part of the same equivalence class (i.e., the target DNN behaves similarly for these inputs).

Neuron coverage is simply the fraction of neurons that are activated across all test inputs. Since all neurons eventually contribute to the output, if we maximize neuron coverage, we should also be maximizing output diversity. For RNN/LSTM models that incorporate loops, intermediate neurons are unrolled to produce a sequence of outputs, with each neuron in an unrolled layer treated as a separate individual neuron for the purpose of coverage computation.

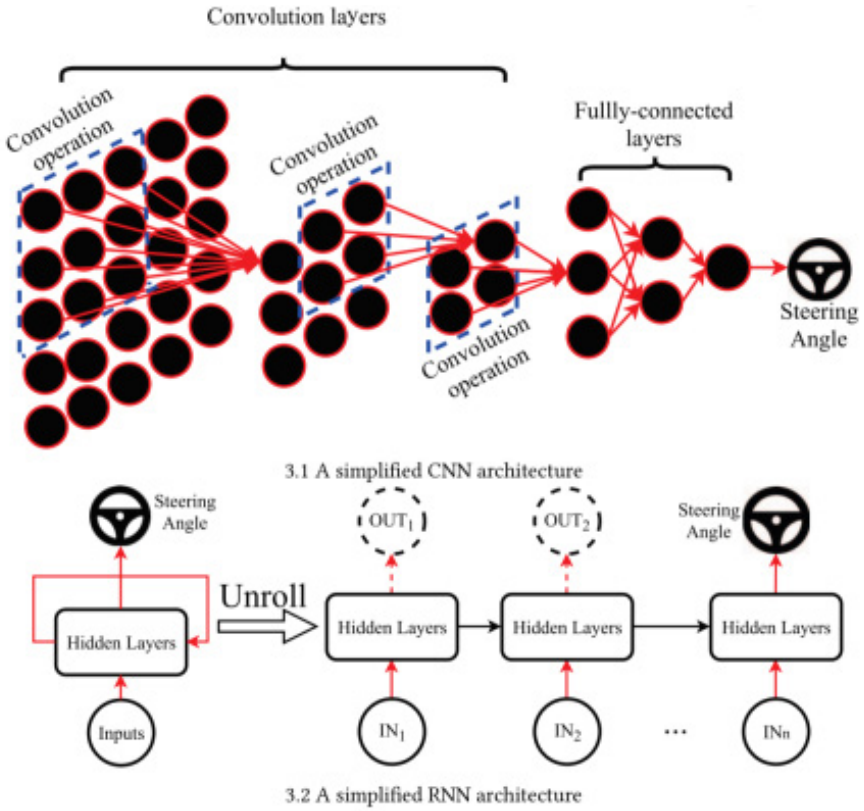


Figure 3: (Upper row) A simplified CNN architecture with a convolution kernel shown on the top-left part of the input image. The same filter (edges with same weights) is then moved across the entire input space, and the dot products are computed between the edge weights and the outputs of the connected neurons. (Lower row) A simplified RNN architecture with loops in its hidden layers. The unrolled version on the right shows how the loop allows a sequence of inputs (*i.e.* images) to be fed to the RNN and the steering angle is predicted based on all those images.

To see whether neuron coverage really is a useful metric in practice, experiments are done with three different DNNs.

Model	Sub-Model	No. of Neurons	Reported MSE	Our MSE
Chauffeur	CNN LSTM	1427 513	0.06	0.06
Rambo	S1(CNN) S2(CNN) S3(CNN)	1625 3801 13473	0.06	0.05
Epoch	CNN	2500	0.08	0.10

† dataset HMB_3.bag [16]

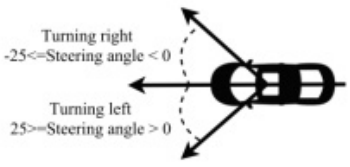


Table 3: (Left) Details of DNNs used to evaluate DeepTest.† (Right) The outputs of the DNNs are the steering angles for a self-driving car heading forward. The Udacity self-driving car has a maximum steering angle of +/- 25 degree.

> ***DeepTest is a system designed to aid in the testing of autonomous-driving models. When used to test three of top performing DNNs from the Udacity self-driving car challenge, it revealed thou- sands of erroneous behaviors, many of which could lead to potentially fatal crashes.***

Table 5: Relation between neuron coverage and test output

Model	Sub-Model	Steering Angle	Steering Direction	
		Spearman Correlation	Wilcoxon Test	Effect size (Cohen's d)
Chauffeur	Overall	-0.10 (***)	left (+ve) > right (-ve) (***)	negligible
	CNN	0.28 (***)	left (+ve) < right (-ve) (***)	negligible
	LSTM	-0.10 (***)	left (+ve) > right (-ve) (***)	negligible
Rambo	Overall	-0.11 (***)	left (+ve) < right (-ve) (***)	negligible
	S1	-0.19 (***)	left (+ve) < right (-ve) (***)	large
	S2	0.10 (***)	not significant	negligible
	S3	-0.11 (***)	not significant	negligible
Epoch	N/A	0.78 (***)	left (+ve) < right (-ve) (***)	small

*** indicates statistical significance with p-value $< 2.2 * 10^{-16}$

The models are fed a variety of inputs, and the neuron coverage, steering angle, and steering-direction (left or right) outputs are recorded. Spearman rank correlation shows a statistically significant correlation between neuron coverage and steering-angle outputs. Different neurons get activated for different outputs, indicating neuron coverage is a good approximation for testing input-output diversity. Steering direction also correlates with neuron coverage. (Table 5)

SYSTEMATICALLY IMPROVING TEST COVERAGE

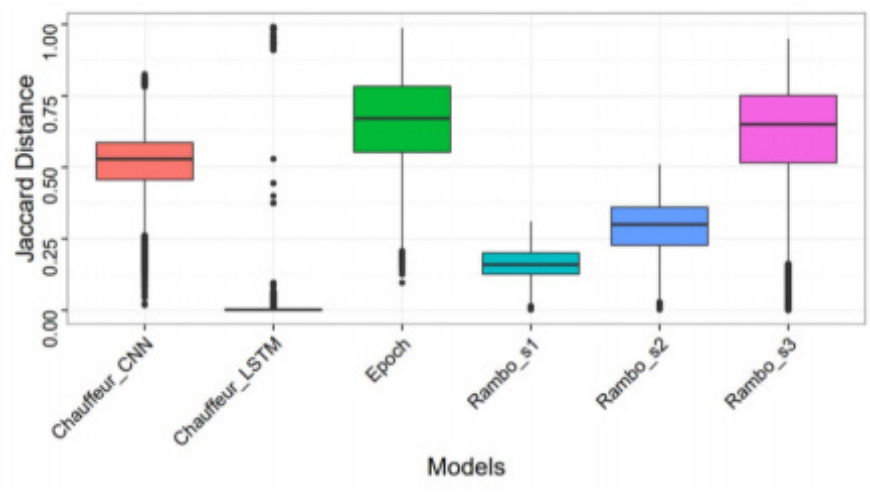
As testers, our goal is to improve the neuron coverage. But how? Synthetic inputs are of limited use, so DeepTest borrows a trick often used to increase diversity in training sets: it applies image transformations to seed images to generate new inputs.

... we investigate nine different realistic image transformations (changing brightness, changing contrast, translation, scaling, horizontal shearing, rotation, blurring, fog effect, and rain effect). These transformations can be classified into three groups: linear, affine, and convolutional. (see table 4)

Starting with 1,000 input images, and applying 70 transformations to each (taken from the core set of transformations, but with varying parameters) generates a set of 70,000 synthetic images. The results show that transforming an image does indeed improve neuron coverage. (see first diagram on next page)

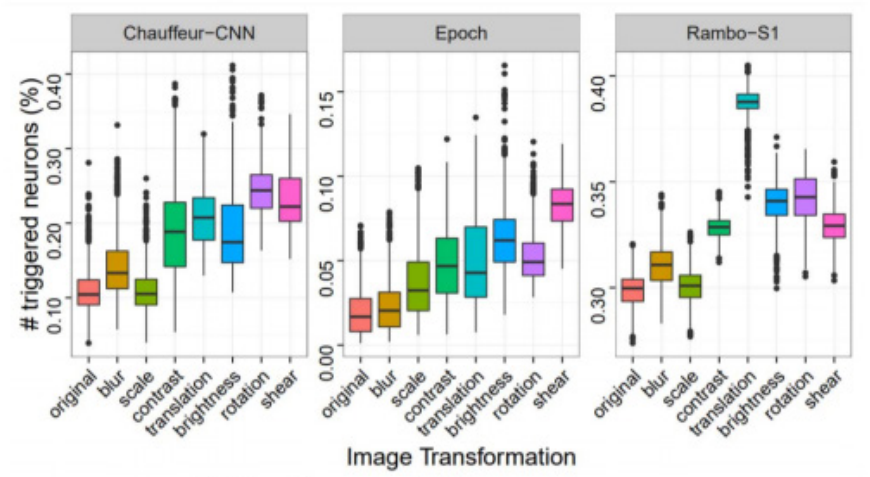
Table 4: Transformations and parameters used by DeepTest for generating synthetic images.

Transformations		Parameters	Parameter ranges
Translation		(t_x, t_y)	(10, 10) to (100, 100) step (10, 10)
Scale		(s_x, s_y)	(1.5, 1.5) to (6, 6) step (0.5, 0.5)
Shear		(s_x, s_y)	(−1.0, 0) to (−0.1, 0) step (0.1, 0)
Rotation		q (degree)	3 to 30 with step 3
Contrast		α (gain)	1.2 to 3.0 with step 0.2
Brightness		β (bias)	10 to 100 with step 10
Blur	**Averaging**	kernel size	3×3, 4×4, 5×5, 6×6
	Gaussian	kernel size	3×3, 5×5, 7×7 , 3×3
	Median	aperture linear size	3, 5
	Bilateral Filter	diameter, sigmaColor, sigmaSpace	9, 75, 75



4.1 Difference in neuron coverage caused by different image transformations

And here's the drill-down by transformation type:

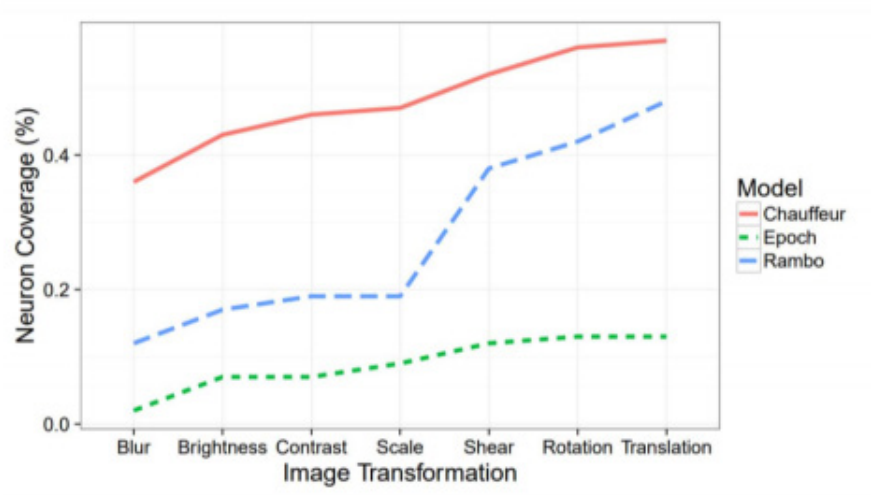


Median Increase in Neuron Coverage

Transformation	Chauffeur (CNN,LSTM)	Epoch	Rambo (S1,S2,S3)
Scale	(1.0,0.0) (0.67%,0%)	39.0** 93%	(2.0*,5.0*,32.0) (0.41%,1%,4%)
Brightness	(100.0**,1.0) (67%,0.2%)	113.0** 269%	(67.0**,104.0**,585.0*) (14%,24%,66%)
Contrast	(120.0**,1.0*) (80%,0.2%)	75.0** 179%	(47.0**,100.0**,159.0) (10%,23%,18%)
Blur	(41.0**,0.0) (28%,0%)	9.0* 21%	(18.0**,23.0**,269.5*) (4%,5%,31%)
Rotation	(199.0**,2.0*) (134%,0.39%)	81.0** 193%	(70.0**,13.0**,786.5*) (14%,3%,89%)
Translation	(147.0**,1.0*) (99%,0.2%)	65.0** 155%	(143.0**,167.0**,2315.5**) (29%,38%,263%)
Shear	(168.0**,1.0*) (113%,0.2%)	167.0** 398%	(48.0**,132.0**,1472.0**) (10%,30%,167%)

All numbers are statistically significant;
Numbers with * and ** have small and large Cohen's D effect.

Figure 5: Neuron coverage per seed image for individual image transformations *w.r.t.* baseline.



4.2 Average cumulative neuron coverage per input image

If one transformation is good, what about applying multiple transformations at the same time to generate a synthetic image? The following chart shows the effect on neuron coverage as we successively apply a sequence of transformations to a given seed image. The results indicate that different image transformations tend to activate different sets of neurons. (see graph above)

Our results demonstrate that different image transformations can be stacked together to further increase neuron coverage. However, the state space of all possible combinations of different transformations is too large to explore exhaustively. We provide a neuron-coverage-guided greedy search technique for efficiently finding combinations of image transformation that result in higher coverage.

The algorithm keeps track of transformations that successfully increase neuron coverage for a given image, and prioritizes those transformations while generating more synthetic images.

These guided transformations increase coverage across all

Algorithm 1: Greedy search for combining image tranformations to increase neuron coverage

```
Input    : Transformations T, Seed images I
Output   : Synthetically generated test images
Variable : S: stack for storing newly generated images
           Tqueue: transformation queue
1
2  Push all seed imgs ∈ I to Stack S
3  genTests = ϕ
4  while S is not empty do
5      img = S.pop()
6      Tqueue = ϕ
7      numFailedTries = 0
8      while numFailedTries ≤ maxFailedTries do
9          if Tqueue is not empty then
10             T1 = Tqueue.dequeue()
11         else
12             Randomly pick transformation T1 from T
13         end
14         Randomly pick parameter P1 for T1
15         Randomly pick transformation T2 from T
16         Randomly pick parameter P2 for T2
17         newImage = ApplyTransforms(image, T1, P1, T2, P2)
18         if covInc(newimage) then
19             Tqueue.enqueue(T1)
20             Tqueue.enqueue(T2)
21             UpdateCoverage()
22             genTest = genTests ∪ newimage S.push(newImage)
23         else
24             numFailedTries = numFailedTries + 1
25         end
26     end
27 end
28 return genTests
```

models, as shown in the following table. Rambo S-3 doesn't improve very much, but note that is was on 98% coverage to start with!

Table 6: Neuron coverage achieved by cumulative and guided transformations applied to 100 seed images.

Model	Baseline	Cumulative Transformation	Guided Generation	% increase of guided w.r.t. Baseline	% increase of guided w.r.t. Cumulative
Chauffeur-CNN	658 (46%)	1,065 (75%)	1,250 (88%)	90%	17%
Epoch	621 (25%)	1034 (41%)	1,266 (51%)	104%	22%
Rambo-S1	710 (44%)	929 (57%)	1,043 (64%)	47%	12%
Rambo-S2	1,146 (30%)	2,210 (58%)	2,676 (70%)	134%	21%
Rambo-S3	13,008 (97%)	13,080 (97%)	13,150 (98%)	1.1%	0.5%

EVALUATING MODEL OUTPUT

We know how to generate inputs that will increase neuron coverage. But how do we know whether or not those inputs reveal a flaw in the network?

The key insight is that even though it is hard to specify the correct behavior of a self-driving car for every transformed image, one can define relationships between the car's behaviors across certain types of transformations.

> ***The algorithm keeps track of transformations that success- fully increase neuron coverage for a given image, and prioritizes those transformations while***

If we change weather or lighting conditions, blur the image, or apply affine transformations with small parameter values (metamorphic relations), we don't expect the steering angle to change significantly. There is a configurable parameter λ that specifies the acceptable deviation from the original image-set results, based on mean squared error versus labeled images in the training set.

As we can see in the figure below, the transformed images produce higher error rates — i.e., they are diverging from the non-transformed output behaviors.

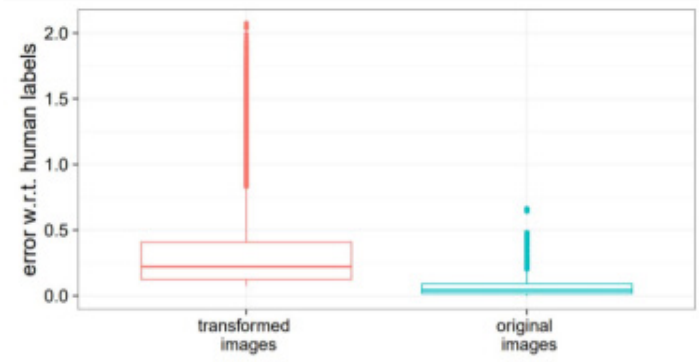


Figure 6: Deviations from the human labels for images that violate the metamorphic relation (see Equation 2) is higher compared to the deviations for original images. Thus, these synthetic images have a high chance to show erroneous behaviors.

RESULTS FROM TESTING AUTONOMOUS-DRIVING DNNS

Using this metamorphic-relation-based test, we can look for differences in model outputs caused by the transformations. DeepTest is able to find quite a lot of them!

Table 7: Number of erroneous behaviors reported b across all tested models at different thresholds

λ (see Eqn. 2)	Simple Tranformation ϵ (see Eqn. 3) 0.01	0.02	0.03	0.04	0.05	Composite Tr Fog	Rain
1	15666	18520	23391	24952	29649	9018	6133
2	4066	5033	6778	7362	9259	6503	2650
3	1396	1741	2414	2627	3376	5452	1483
4	501	642	965	1064	4884	4884	997
5	95	171	**330**	382	641	**4448**	**741**
6	49	85	185	210	359	4063	516
7	13	24	89	105	189	3732	287

Here are some sample images from the failing tests. You can see more at https://deeplearningtest.github.io/deepTest/.

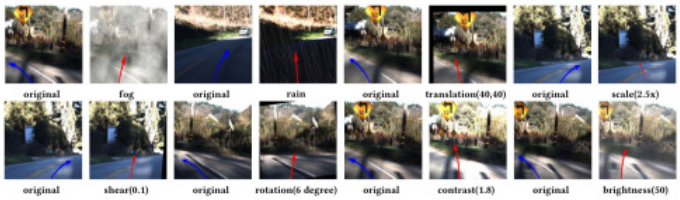


Figure 7: Sample images showing erroneous behaviors detected by DeepTest using synthetic images. For original images the arrows are marked in blue, while for the synthetic images they are marked in red. More such samples can be viewed at https://deeplearningtest.github.io/deepTest/.

Manual checking reveals two false positives where DeepTest reports erroneous behaviors but the outputs (as assessed by the authors) actually are safe.

Model	Simple Transformation	Guided	Rain	Fog	Total
Epoch	14	0	0	0	14
Chauffeur	5	3	12	6	26
Rambo	8	43	11	28	90
Total	27	46	23	34	130

original translation(50,50), epoch original shear(0.4), rambo

Figure 8: Sample false positives produced by DeepTest for $\lambda = 5$, $\epsilon = 0.03$

FIXING TEST FAILURES

Retraining the DNNs with some of the synthetic images generated by DeepTest makes them more robust, as shown in the table below.

Table 9: Improvement in MSE after retraining of Epoch model with synthetic tests generated by DeepTest

Test set	Original MSE	Retrained MSE
original images	0.10	0.09
with fog	0.18	0.10
with rain	0.13	0.07

WHAT ABOUT YOUR DNN?

We use domain-specific metamorphic relations to find erroneous behaviors of the DNN without detailed specification. DeepTest can be easily adapted to test other DNN-based systems by customizing the transformations and metamorphic relations. We believe DeepTest is an important first step towards building robust DNN-based systems.

HERE IS WHAT WE'VE COVERED IN THE PREVIOUS ISSUES

A DIRTY DOZEN: TWELVE COMMON METRIC INTERPRETATION PITFALLS IN ONLINE CONTROLLED EXPERIMENTS

SEVEN RULES OF THUMB FOR WEB SITE EXPERIMENTERS

THE EVOLUTION OF CONTINUOUS EXPERIMENTATION IN SOFTWARE PRODUCT DEVELOPMENT

GOOGLE VIZIER: A SERVICE FOR BLACK-BOX OPTIMISATION

TFX: A TENSORFLOW-BASED PRODUCTION-SCALE MACHINE-LEARNING PLATFORM

GRAY FAILURE: THE ACHILLES' HEEL OF CLOUD-SCALE SYSTEMS

TRAJECTORY RECOVERY FROM ASH: USER PRIVACY IS NOT PRESERVED IN AGGREGATED MOBILITY DATA

IOT GOES NUCLEAR: CREATING A ZIGBEE CHAIN REACTION

SYSTEM PROGRAMMING IN RUST: BEYOND SAFETY

MOSAIC: PROCESSING A TRILLION-EDGE GRAPH ON A SINGLE MACHINE