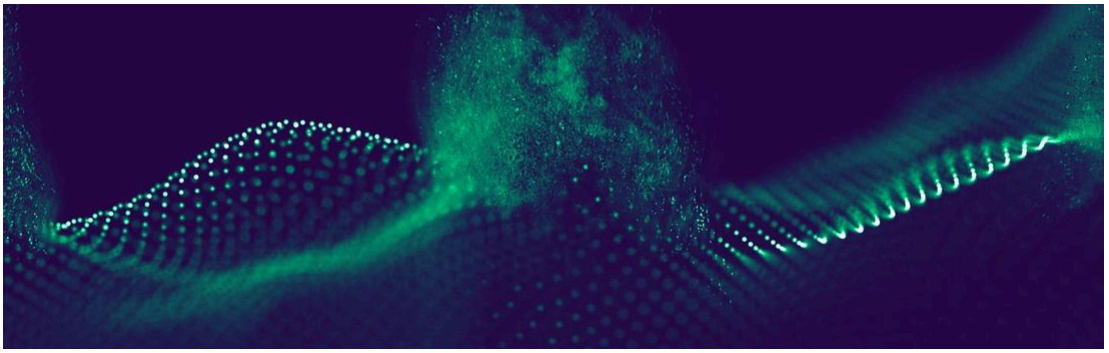
5 Key Challenges in Machine Learning Development

Originally published at https://www.iflexion.com/blog/machine-learning-challenges

Published: September 22, 2020 | By Martin Anderson



The phrase *'Insanity is doing the same thing over and over again and expecting different results'* has found a foothold in popular wisdom in recent decades, even if its provenance remains disputed. Such stubborn behavior confounds scientific method, exposes an immature tendency in human psychology, and accords with our own experience of achieving change and progress. It applies everywhere.

Except in machine learning.

Machine learning has the opposite problem, in that neural networks cannot *exactly* reproduce the efficacy of previous results even where all the tightly-controlled variables are the same: the same data, the same hardware, the same methodologies.

When the need arises to migrate to new software versions, better loss functions, upgraded hardware, revised/amended data, or to add or reduce model complexity, precise reproducibility drops even further — and all of those circumstances are frequent and inevitable. In this article on the challenges of AI software development, we'll take a look at five key areas in setting up a machine learning model where minor changes can yield critical differences in usability and performance.

Herding Cats in a Neural Network

Industry faith (and ongoing investment) in new technologies depends on reproducibility and on explicable and predictable processes. Where a process is successful but occult, it's expected to be a proprietary technology, such as the profitable Google search algorithm.

By contrast, nearly all of machine learning frameworks are open-source and accessible to all. How is it possible, given this level of transparency, that the AI and machine learning sectors struggle against a popular perception that they are 'black-box' technologies? Why is it so difficult[1] to industrialize complex reproducible outcomes from machine learning models? Why is extracting core truths from big data so annoyingly like herding cats?

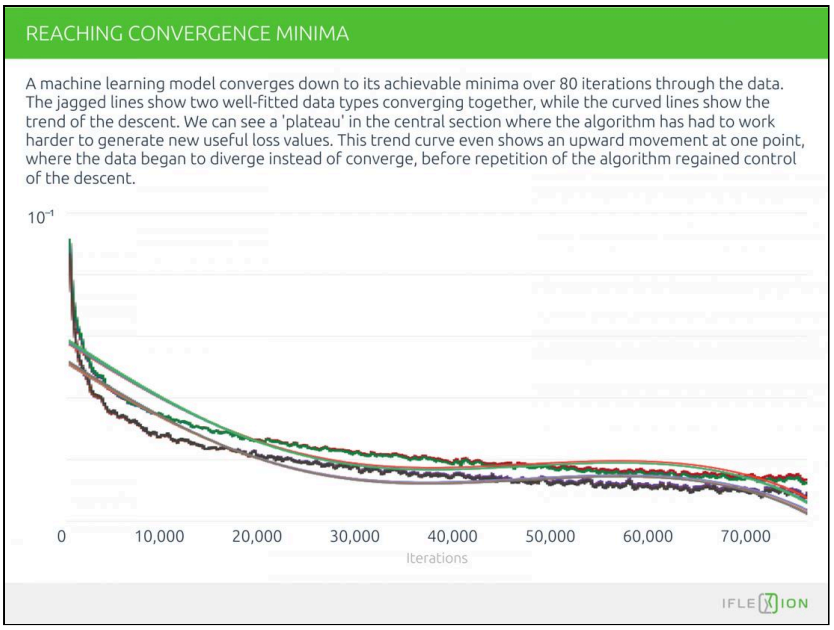
Controlling Convergence

The goal in the development of a machine learning model is to identify central relationships and potential transformations in large amounts of data, in a manner that enables it to repeat the process later on a similarly structured but different set of data.

To achieve this, the model must traverse large amounts of input training data and establish the 'neural pathways' through which similar information will travel in future sessions (hopefully in a profitable or otherwise beneficial way). When the model has understood and established the innate relationships in the data, it has achieved *convergence*.

In mathematics, a 'convergent sequence' is a formula that will ultimately resolve to a fixed point, or *minima*, which might be a specific outcome or a narrow gamut of possible outcomes that will not become any narrower with further processing of the data.

So a convergent algorithm is a reductionist device designed to determine the most useful and generalized outcome from a large volume of possible outcomes, by systematically applying a formula and rejecting what it perceives to be the least accurate results *in each iteration*.



It is still difficult to industrialize complex reproducible outcomes from machine learning models. Why?

1: Achieving Performant Weights in Machine Learning Algorithms

To accomplish convergence, the algorithm needs to decide in advance how 'ruthless' it will be in rejecting results from each iteration. In the case of machine learning, it's important that this criterion for rejection becomes more and more fine-grained as the process continues.

By way of analogy, a traditional carpenter's first tool in the creation of a table might be a crude axe, while their last tools could include the finest-grade sandpaper and the most delicate of engraving instruments. If the carpenter was to exclusively use either approach, the table would either be destroyed in a blizzard of woodchips in the first hour or else take several years to make.

In machine learning models, these parameters (or 'limiters') are called *weights* and need constant adjustment and refinement as the model evolves. Depending on how the weights are set, and how the model's bias is set to influence the weights, one risks to create an algorithm that either 'shreds the wood' uselessly or else only ever learns how to make one specific table, rather than a range of tables (see 'Over-Fitting and Under-Fitting' later).

Back-Propagation

In a typical neural network, a back-propagation algorithm examines the results of each iteration in the machine learning model's evaluation of the training data and refines or alters the weights based on recent performance indicators, so that the descent towards the local minima (the objective of the model) does not get interrupted.

This trajectory towards the minima is the 'gradient descent' of the model — a (hopefully) consistent inclination downwards from a high to a low loss value, and ultimately to convergence, where the model has assimilated the essential properties of the training data and is ready to apply it fruitfully to new data of a similar type.

2: Choosing the Right Loss Function

The loss function (also known as the Cost Function) chosen for a machine learning model is a key determining factor in how the model will converge and ultimately perform in a later deployment.

In itself, loss is a number that indicates how far the neural network strayed from its goal while processing the latest iteration of the data. The lower the number, the nearer the model is to convergence — the point at which the essential features of the training data have been assimilated and integrated into a practical and exploitable template for future analyses of new data input. At the start of a round of training, initial average loss numbers might hover, for instance, around the 0.9000 mark, descending on a curve to a more useful 0.0100 loss value. In most cases the loss values will *plummet* initially, burning through the 0.9000>0.3000 range before slowing down noticeably.

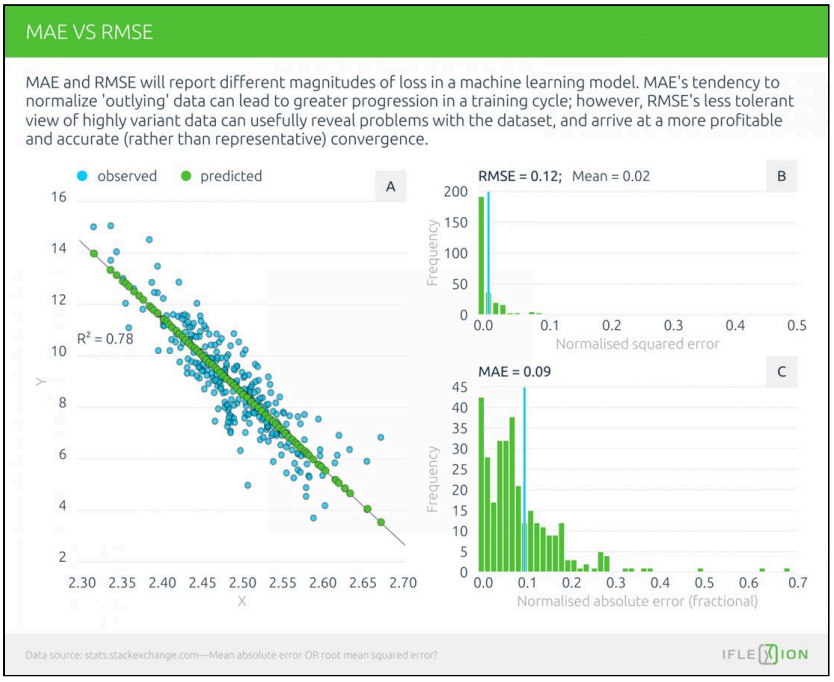
This deceleration occurs because each loss drop is harder to achieve, with the model's descent incrementally slowing towards a usable convergence, known as the 'global optimum'.

Assuming the data itself is in good order and no further data cleaning is necessary, the limit, rate and clarity of this descent will be determined to a great extent by the loss function chosen for the model. There are many available[2], even within the narrower ambit of a sub-sector of machine learning (e.g., natural language processing or computer vision), and the applicability of any of them will be determined by a number of factors.

MAE vs. RMSE

For instance, where the training data is less consistent, the Mean Absolute Error (MAE) loss function will maintain consistency in the face of 'outliers' — data points that skew wildly away from the average values of the data set.

On the other hand, the Root Mean Squared Error (RMSE) loss algorithm gives a higher weight to large errors[3], which can help to determine whether or not the input data is consistent enough within itself to converge usefully. Where great deviations from the norm would damage the integrity or usefulness of the model, RMSE can be a useful investigative tool, as well as the right choice for a production model (though MAE is more frequently used).



3: Controlling Learning Rate Schedules

A machine learning model is configured to learn at a certain speed initially. Much as an artist might quickly put in broad strokes on a canvas, the learning rate *annealing* approach (also known as Linear or Exponential Time-Based Decay) suggests that initial speed should be fairly high. At this stage, the model only has to identify the greater tendencies or general 'shape' of the potential relationships and transformations.

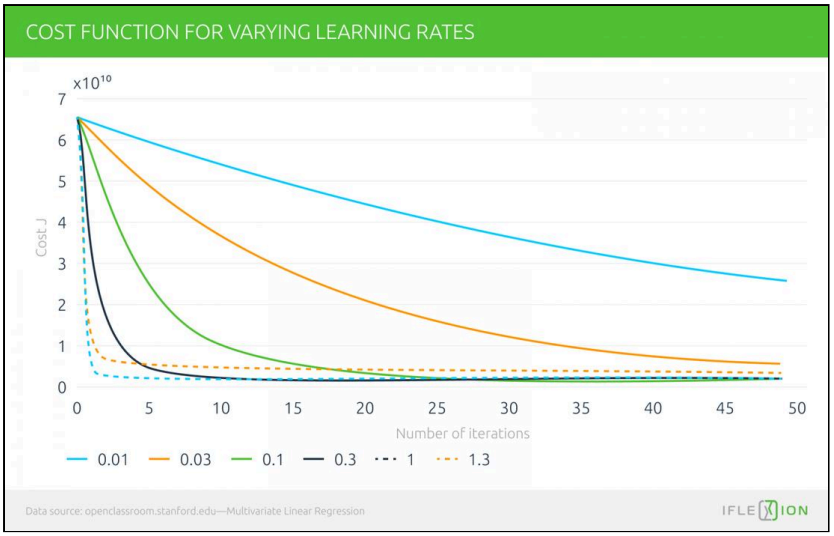
Much as the weights must be adjusted throughout the learning process (as we have seen), the learning speed should also decrease so that the processing power is gradually directed in a more concentrated way at the emerging relationships in the data.

Establishing an Initial Learn Rate

Though it entails some extra initial effort and time-cost, a starting learning rate can be determined by a process of elimination, raising the learning rate from low values to higher values several times on a limited set of iterations until divergence (failure to learn) occurs, and then setting the starting learn rate a point or two below that value.

Thereafter, there are no absolute rules about the best way to adjust the learning speed in a training session; if the speed is lowered too much (or too early in the data processing cycle), the model can become 'stuck', mistaking a small local resolution for a useful general convergence.

Alternately, if the learning rate is left at a faster rate for too long, the model can race to a hasty conclusion, having converged completely but unsuccessfully and missed the subtle inter-relationships that it was seeking in the data, because the larger areas of generalization got resolved too fast, and the data 'ran out of road'.



Various machine learning libraries offer methods to implement a learning rate schedule, which automates the learning rate variation according to time passed, rather than perceived drops in the loss value (which is how weights are altered through back-propagation). For instance, Keras has a time-based learning rate scheduler[4].

An alternate approach is a drop-based learning rate schedule, which decreases the learning rate based not on time passed but on iterations achieved. This too can be implemented in Keras[5].

A learning rate schedule can also be implemented manually, based on judgement and previous experience.

There are pitfalls in programming a machine learning model to learn either to fast or too slow.

4: Coping with Innate Randomness in a Machine Learning Model

The ultimate aim of a learning rate schedule, as with all other parameters involved in the configuration of a machine learning model, is to train a model that can consistently and successfully process the same type of data in future training sessions, obtaining a useful convergence each time.

In practice, it is not only impossible to obtain this over data sets that differ from the original training one, but it is usually not possible to obtain *exactly the same result* twice from the same data, even when using the same hardware and model configuration[6].

Data from the training set is never fed into the model in the same sequence in the course of development for any two separate models, because stochastic machine learning algorithms rely on randomness[7] to access and develop different areas of the data.

Minor Changes Can Have a Large Downstream Influence

To telescope the issue, consider that in many cases an airplane deviating from its set course by one degree at the start of a six-hour journey is likely to end up in a different *country* than its intended destination. Though a machine learning model will ultimately re-orient its approximate path to consider the entirety of the data set (rather than fixating on the random characteristics of the first data it samples), a 'reproduced' training session is nonetheless always working from a slightly different set of initial assumptions, even where the training data is identical to previous occasions.

Furthermore, the less consistent the data, the more of a downstream effect this randomness is likely to have on the way the model develops[8].

Additionally, model evaluation and prediction can be notably affected by changes in the production environment, such as updated machine learning libraries and variations in the way that different CPUs and GPUs may approach differences in rounding errors.

Cleaning the Data vs. Rethinking the Approach

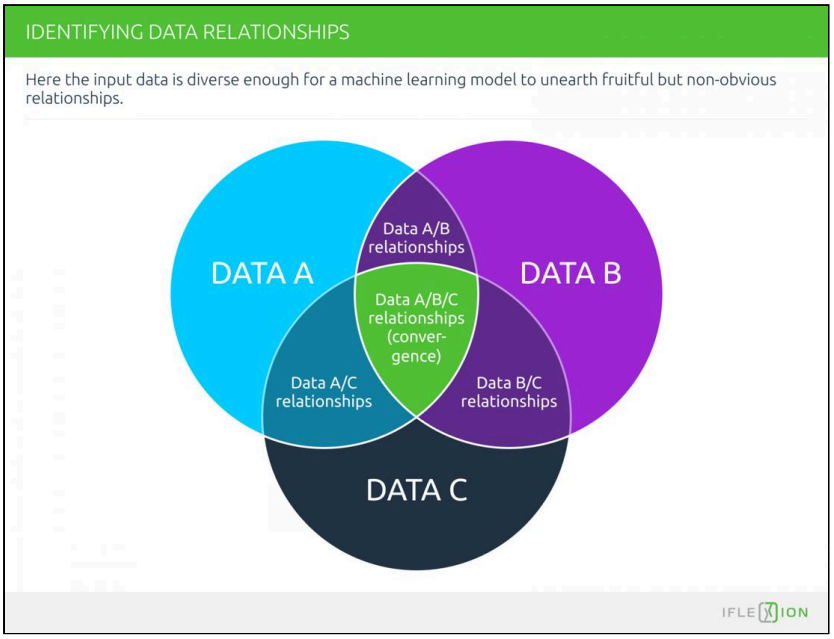
There is a limitation to the extent to which this challenge can be addressed by cleaning and labelling consistent data. If a neural network cannot reach exactly the same configuration twice from identical (training) data, subsequent data runs will inevitably not produce *precisely* the same quality of transformations as the first.

Furthermore, if a series of data sets *could* achieve enough homogeneity to avoid this pitfall, there is arguably nothing useful that a machine learning system could deduce from them (see #5 below).

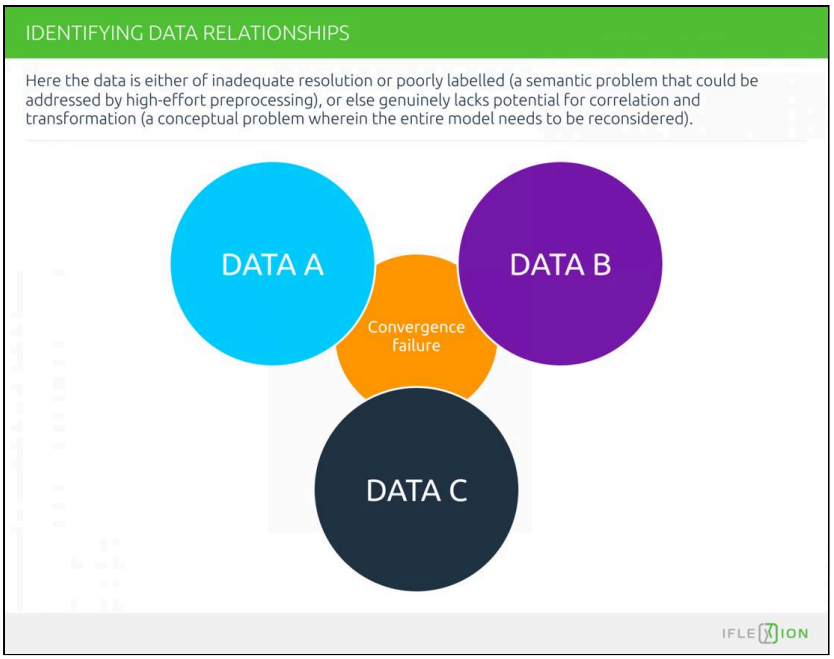
5: Achieving 'Useful Dissonance' in a Training Data Set

Though it is beyond the scope of this article to address the huge subject of data preprocessing in any detail, we also need to consider the value of maintaining a tension between 'dirty' and 'clean' data, according to our intended aims for the model.

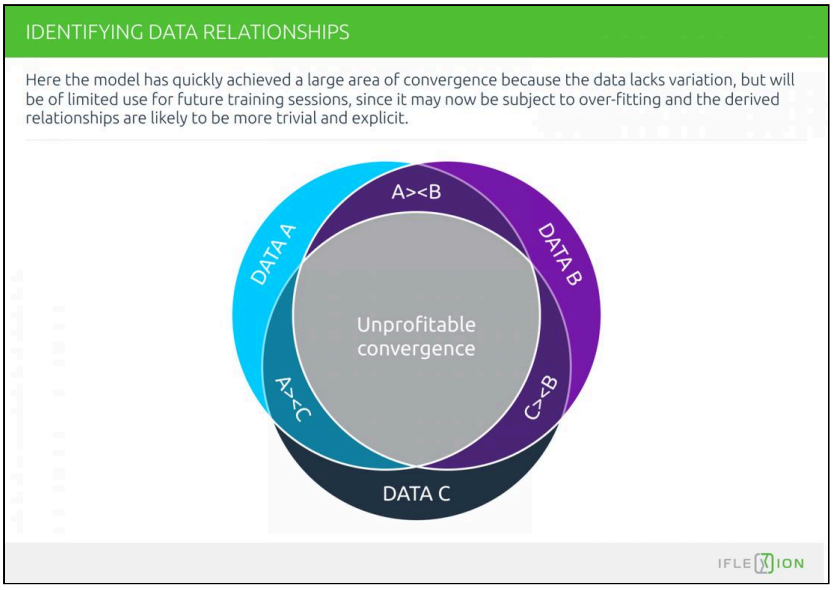
If we consider that the objective of a machine learning algorithm is to reveal hidden correlations and potential transformations between a collection of different data points, we can visualize this as the apex of a Venn diagram, where there is enough dissonance between the data points to make any new relationships that the machine learning model might identify as insightful and exploitable, rather than obvious:



Where the diversity of the data is much greater, so that there are no apparent commonalities between them, any relationships the machine learning model forms are likely to be specious, non-reproducible, and of low value:



Alternately, with inadequate variation in a data set, we may achieve a facile convergence that is neither useful nor resilient, because the relationships were likely quite clear to begin with, and defining them was so easy to achieve that the model did not form 'neural pathways' flexible enough to draw useful conclusions from subsequent, more challenging data runs.



Though there are research initiatives that hope to reduce or eliminate the burden of data preprocessing[9], the choice of data and the extent of preprocessing has a critical influence on the success of a machine learning model. It can be extremely difficult for AI engineers to distinguish between a model that needs structural revision, data that needs additional processing, or assumptions about the data's potential that may need reevaluation.

Over-Fitting and Under-Fitting

Over-fitting often occurs when a data set is trained so intensively by the machine learning model that it begins to evaluate the data's 'noise' (rather than just its central form) as a critical characteristic. It also occurs when an overly complex or capacious model trains a relatively undemanding data set.

An over-fit and under-generalized model is easy to recognize, as it performs very well on the original data but very poorly on subsequent data sets of a similar type. Overfitting can be addressed by controlling weight decay in Keras[10] and similar frameworks.

Under-fitting can occur when the neural network model is not complex or capacious enough to accommodate the richness of the input data. Though it is easily solved by improving the complexity and capacity of the model, it is harder to identify as the cause of convergence failure, since similar negative results can be obtained by poorly labelled or badly-processed data, or else by conceptual issues regarding what the data is capable of achieving in a machine learning model.