The Technological Elements of Artificial Intelligence

Matt Taddy, *Chicago Booth and Microsoft*

1 Introduction

We have seen in the past decade a sharp increase in the extent that companies use data to optimize their businesses. Variously called the 'Big Data' or 'Data Science' revolution, this has been characterized by massive amounts of data, including unstructured and nontraditional data like text and images, and the use of fast and flexible Machine Learning (ML) algorithms in analysis. With recent improvements in Deep Neural Networks (DNNs) and related methods, application of high-performance ML algorithms has become more automatic and robust to different data scenarios. That has led to the rapid rise of an Artificial Intelligence (AI) that works by combining many ML algorithms together – each targeting a straightforward prediction task – to solve complex problems.

In this chapter, we will define a framework for thinking about the ingredients of this new ML-driven AI. Having an understanding of the pieces that make up these systems and how they fit together is important for those who will be building businesses around this technology. Those studying the economics of AI can use these definitions to remove ambiguity from the conversation on AI's projected productivity impacts and data requirements. Finally, this framework should help clarify the role for AI in the practice of modern business analytics and economic measurement.

2 What is AI?

In Figure 1, we show a breakdown of AI into three major and essential pieces. A full end-to-end AI solution – i.e., what Microsoft calls a *System of Intelligence* – is able to ingest human-level knowledge (e.g., via machine reading and computer vision) and use this information to automate and accelerate tasks that were previously only performed by humans. It is necessary here to have a well-defined task structure to engineer against, and in a business setting this structure is provided by business and economic domain expertise. You need a massive bank of data to get the system up and running, and a strategy to continue generating data so that the system can respond and learn. And finally, you need Machine Learning routines that can detect patterns in and make predictions from the unstructured data. This section will work through each of these pillars, and in later sections we dive in detail into Deep Learning models, their optimization, and data generation.

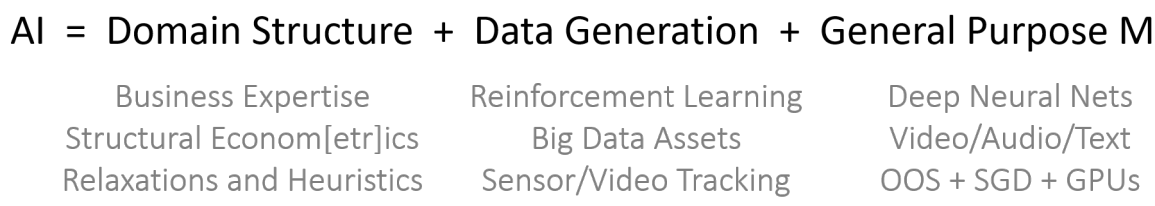

Figure 1: AI systems are self-training structures of ML predictors that automate and accelerate human tasks.

Notice that we are explicitly separating ML from AI here. This is important: these are different but often confused technologies. ML can do fantastic things, but it is basically limited to predicting a future that looks mostly like the past. These are tools for pattern recognition. In contrast, an AI system is able to solve complex problems that have been previously reserved for humans. It does this by breaking these problems into a bunch of simple prediction tasks, each of which can be attacked by a 'dumb' ML algorithm. AI *uses* instances of Machine Learning as components of the larger system. These ML instances need to be organized within a structure defined by domain knowledge, and they need to be fed data that helps them complete their allotted prediction tasks.

This is not to down-weight the importance of ML in AI. In contrast to earlier attempts at AI, the current instance of AI is *ML-driven*. ML algorithms are implanted in every aspect of AI, and below we describe the evolution of Machine Learning towards status as a general purpose technology. This evolution is the main driver behind the current rise of AI. However, ML algorithms are building blocks of AI within a larger context.

To make these ideas concrete, consider an example AI system from the Microsoft-owned company Maluuba that was designed to play (and win!) the video game Ms Pac-Man on Atari.[43] The system is illustrated in Figure 2. The player moves Ms Pac-Man on this game 'board', gaining rewards for eating pellets while making sure to avoid getting eaten by one of the adversarial 'ghosts'. The Maluuba researchers were able to build a system that learned how to master the game, achieving the highest possible score and surpassing human performance.

A common misunderstanding of AI imagines that, in a system like Maluuba's, the player of the game *is* a Deep Neural Network. That is, the system works by swapping out the human joy-stick operator for an artificial DNN 'brain'. That's not how it works. Instead of a single DNN that is tied to the Ms Pac-Man avatar (which is how the human player experiences the game), the Maluuba system is broken down into 163 component ML tasks. As illustrated on the right panel of Figure 2, the engineers have assigned a distinct DNN routine to each cell of the board. In addition, they have DNNs that track the game characters: the ghosts and, of course, Ms Pac-Man herself. The direction that the AI system sends Ms Pac-Man at any point in the game is then chosen through consideration of the advice from each of these ML components. Recommendations from the components that are close to Ms Pac-Man's current board position are weighted more strongly than those of currently remote locations. Hence, you can think of the ML algorithm assigned to each square on the board as having a simple task to solve: when Ms Pac-Man crosses over this location, which direction should she go next?

Learning to play a video or board game is a standard way for AI firms to demonstrate their current capabilities. The Google DeepMind system AlphaGo[37], which was constructed to play the fantastically complex board-game 'Go', is the most prominent of such demonstrations. The system was able to surpass human capability, beating the world champion, Lee Sedol, 4 matches

Figure 2: Screen-shots of the Maluuba system playing Ms Pac-Man. On the left, you see the game board. On the right, the authors have assigned arrows showing the current direction for Ms Pac-Man that is advised by different locations on the board, each corresponding to a distinct Deep Neural Network.

to 1 at a live-broadcast event in Seoul, South Korea, in March 2016. Just as Maluuba's system broke Ms Pac-Man into a number of composite tasks, AlphaGo succeeded by breaking Go into an even larger number of ML problems: 'value networks' that evaluate different board positions and 'policy networks' that recommend moves. The key point here is that while the composite ML tasks can be attacked with relatively generic DNNs, the full combined system is constructed in a way that is highly specialized to the structure of the problem at hand.

In Figure 1, the first listed pillar of AI is *domain structure*. This is the structure that allows you to break a complex problem into composite tasks that can be solved with ML. The reason that AI firms choose to work with games is that such structure is explicit: the rules of the game are codified. This exposes the massive gap between playing games and a system that could replace humans in a real-world business application. To deal with the real world, you need to have a theory as to the rules of the relevant game. For example, if you want to build a system that can communicate with customers you might proceed by mapping out customer desires and intents in such a way that allows different dialog-generating ML routines for each. Or, for any AI system that deals with marketing and prices in a retail environment, you need to be able to use the structure of an economic demand system to forecast how changing the price on a single item (which might, say, be the job of a single DNN) will affect optimal prices for other products and behavior of your consumers (who might themselves be modeled with DNNs).

The success or failure of an AI system is defined in a specific *context*, and you need to use the structure of that context to guide the architecture of your AI. This is a crucial point for businesses hoping to leverage AI and economists looking to predict its impact. As we will detail below, Machine Learning in its current form has become a *general purpose technology*.[6] These tools are going to get cheaper and faster over time, due to innovations in the ML itself and above and below in the AI technology stack (e.g., improved software connectors for business systems above, and improved computing hardware like GPUs below). ML has the potential to become a cloud computing commodity.[1] In contrast, the domain knowledge necessary to combine ML components into an end-to-end AI solution will not be commoditized. Those who have expertise that can break

[1] Amazon, Microsoft, and Google are all starting to offer basic ML capabilities like transcription and image classification as part of their cloud services. The prices for these services are low and mostly matched across providers.

complex human business problems into ML-solvable components will succeed in building the next generation of business AI, that which can do more than just play games.

In many of these scenarios, Social Science will have a role to play. Science is about putting structure and theory around phenomena that are observationally incredibly complex. Economics, as the Social Science closest to business, will often be relied upon to provide the rules for business AI. And since ML-driven AI relies upon measuring rewards and parameters inside its context, *econometrics* will play a key role in bridging between the assumed system and the data signals used for feedback and learning. The work will not translate directly. We need to build systems that allow for a certain margin of error in the ML algorithms. Those economic theories that apply for only a very narrow set of conditions – e.g., at a knife's edge equilibrium – will be too unstable for AI. This is why we mention relaxations and heuristics in Figure 1. There is an exciting future here where economists can contribute to AI engineering, and both AI and Economics advance as we learn what recipes do or do not work for Business AI.

Beyond ML and domain structure, the third pillar of AI in Figure 1 is *data generation*. I'm using the term 'generation' here, instead of a more passive term like 'collection', to highlight that AI systems require an active strategy to keep a steady stream of new and useful information flowing into the composite learning algorithms. In most AI applications there will be two general classes of data: fixed-size data assets that can be used to train the models for generic tasks, and data that is actively generated by the system as it experiments and improves performance. For example, in learning how to play Ms Pac-Man the models could be initialized on a bank of data recording how humans have played the game. This is the fixed size data asset. Then this initialized system starts to *play* the game of Ms Pac-Man. Recalling that the system is broken into a number of ML components, as more games are played each component is able to experiment with possible moves in different scenarios. Since all of this automated, the system can iterate through a massive number of games and quickly accumulate a wealth of experience.

For business applications, we should not underestimate the advantage of having large data assets to initialize AI systems. Unlike board or video games, real-world systems need to be able to be able to interpret a variety of extremely subtle signals. For example, any system that interacts with human dialog must be able to understand the general domain language before it can deal with specific problems. For this reason, firms that have large banks of human interaction data (e.g., social media or a search engine) have a large technological advantage in conversational AI systems. However, this data just gets you started. The context-specific learning starts happening when, after this 'warm start', the system begins interacting with real-world business events.

The general framework of ML algorithms actively choosing the data that they consume is referred to as *Reinforcement Learning* (RL).[2] It is a hugely important aspect of ML-driven AI, and we have a dedicated section on the topic below. In some narrow and highly-structured scenarios, researchers have build 'zero-shot' learning systems where the AI is able to achieve high performance after stating without any static training data. For example, in subsequent research, Google DeepMind has developed the AlphaGoZero[38] system that uses zero-shot learning replicate their earlier AlphaGo success. Noting that the RL is happening on the level of individual ML tasks, we can update our description of AI as being composed of many RL-driven ML components.

As a complement to the work on reinforcement learning, there is a lot of research activity

[2]This is an old concept in statistics. In previous iterations, parts of reinforcement learning have been referred to as the sequential design of experiments, active learning, and Bayesian optimization.

around AI systems that can simulate 'data' to appear as though it came from a real-world source. This has the potential to accelerate system training, replicating the success that the field has had with video and board games where experimentation is virtually costless (just play the game, nobody loses money or gets hurt). Generative Adversarial Networks[10] (GANs) are schemes where one DNN is simulating data and another is attempting to discern which data is real and which is simulated. For example, in an image-tagging application one network will generate captions for the image while the other network attempts to discern which captions are human vs machine generated. If this scheme works well enough, then you can build an image tagger while minimizing the number of dumb captions you need to show humans while training.

And finally, AI is pushing into physical spaces. For example, the Amazon Go concept promises a frictionless shopping-checkout experience where cameras and sensors determine what you've taken from the shelves and charge you accordingly. These systems are as data intensive as any other AI application, but they have the added need to translate information from a physical to a digital space. They need to be able to recognize and track both objects and individuals. Current implementations appear to rely on a combination of object-based data sources, via sensor and device networks (i.e., the IoT or 'Internet of Things'), and video data from surveillance cameras. The sensor data has the advantage that it is well structured and tied to objects, but the video data has the flexibility to look in places and at objects that you didn't know to tag in advance. As computer vision technology advances, and as the camera hardware adapts and decreases in cost, we should see a shift in emphasis towards unstructured video data. We've seen similar patterns in AI development, e.g., as use of raw conversation logs increases with improved machine reading capability. This is the progress of ML-driven AI towards general purpose forms.

3 General Purpose Machine Learning

The piece of AI that gets the most publicity – so much so that it is often confused with all of AI – is *general purpose* Machine Learning. Regardless of this slight overemphasis, it is clear that the recent rise of Deep Neural Networks (DNNs; see the Deep Learning section below) is a main driver behind growth in AI. These DNNs have the ability to learn patterns in speech, image, and video data (as well as in more traditional structured data) faster, and more automatically, than ever before. They provide new ML capabilities and have completely changed the work-flow of an ML engineer. However, this technology should be understood as a rapid evolution of existing ML capabilities rather than as a completely new object.

Machine Learning is the field that thinks about how to automatically build robust predictions from complex data. It is closely related to modern Statistics, and indeed many of the best ideas in ML have come from Statisticians (the lasso, trees, forests, etc). But whereas statisticians have often focused *model inference* – on understanding the parameters of their models (e.g., testing on individual coefficients in a regression) – the ML community has been more focused on the single goal of maximizing *predictive performance*. The entire field of ML is calibrated against 'out-of-sample' experiments that evaluate how well a model trained on one dataset will predict new data. And while there is a recent push to build more transparency into Machine Learning, wise ML practitioners will avoid assigning structural meaning to the parameters of their fitted models. These models are black boxes whose purpose is to do a good job in predicting a future that follows the same patterns as in past data.

Prediction is easier than model inference. This has allowed the ML community to quickly push forward and work with larger and more complex data. It also facilitated a focus on automation: developing algorithms that will work on a variety of different types of data with little or no tuning required. We've seen an explosion of general purpose ML tools in the past decade – tools that can be deployed on messy data and automatically tuned for optimal predictive performance.

The specific ML techniques used include high-dimensional ℓ_1 regularized regression (Lasso), tree algorithms and ensembles of trees (e.g., Random Forests), and Neural Networks. These techniques have found application in business problems, under such labels as 'Data Mining' and, more recently, 'Predictive Analytics'. Driven by the fact that many policy and business questions require more than just prediction, practitioners have added an emphasis on inference and incorporated ideas from Statistics. Their work, combined with the demands and abundance of Big Data, coalesced together to form the loosely defined field of Data Science. More recently, as the field matures and as people recognize that not everything can be explicitly A/B tested, Data Scientists have discovered the importance of careful causal analysis. One of the most currently active areas of Data Science is combining ML tools with the sort of counter-factual inference that econometricians have long studied, hence now merging the ML and Statistics material with the work of Economists. See, e.g., Athey and Imbens [3], Hartford et al. [13], and the survey in Athey [2].

The push of ML into the general area of Business Analytics has allowed companies to gain insight from high dimensional and unstructured data. This is only possible because the ML tools and recipes have become robust and usable enough that they can be deployed by non-experts in Computer Science or Statistics. That is, they can be used by people with a variety of quantitative backgrounds who have domain knowledge for their business use-case. Similarly, the tools can be used by Economists and other Social Scientists to bring new data to bear on scientifically compelling research questions. Again: the general usability of these tools has driven their adoption across disciplines. They come packaged as quality software and include validation routines that allow the user to observe how well their fitted models will perform in future prediction tasks.

The latest generation of ML algorithms, especially the Deep Learning technology that has exploded since around 2012[23], has increased the level of *automation* in the process of fitting and applying prediction models. This new class of ML is the *general purpose ML* (GPML) that we reference in the rightmost pillar of Figure 1. The first component of GPML is Deep Neural Networks: models made up of *layers* of nonlinear transformation *node* functions, where the output of each layer becomes input to the next layer in the network. We will describe DNNs in more detail in our Deep Learning section below, but for now it suffices to say that they make it faster and easier than ever before to find patterns in unstructured data. They are also highly modular. You can take a layer that is optimized for one type of data (e.g., images) and combine it with other layers for other types of data (e.g., text). You can also use layers that have been pre-trained on one dataset (e.g., generic images) as components in a more specialized model (e.g., a specific recognition task).

Specialized DNN architectures are responsible for the key GPML capability of working on human-level data: video, audio, and text. This is essential for AI because it allows these systems to be installed on top of the same sources of knowledge that humans are able to digest. You don't need to create a new data-base system (or have an existing standard form) to feed the AI; rather, the AI can live on top of the chaos of information generated through business functions. This capability helps to illustrate why the new AI, based on GPML, is so much more promising than previous attempts at AI. Classical AI relied on hand-specified logic rules to mimic how a rational human might approach a given problem.[14] This approach is sometimes nostalgically referred

to as GOFAI, or 'good old-fashioned AI'. The problem with GOFAI is obvious: solving human problems with logic rules requires an impossibly complex cataloging of all possible scenarios and actions. Even for systems able to learn from structured data, the need to have an explicit and detailed data schema means that the system designer must to know in advance how to translate complex human tasks into deterministic algorithms.

The new AI doesn't have this limitation. For example, consider the problem of creating a virtual agent that can answer customer questions (e.g., 'why won't my computer start?'). A GOFAI system would be based on hand-coded dialog trees: if a user says X, answer Y, etc. To install the system, you'd need to have human engineers understand and explicitly code for all of the main customer issues. In contrast, the new ML-driven AI can simply ingest all of your existing customer-support logs and learn to replicate how human agents have answered customer questions in the past. The ML allows your system to infer support patterns from the human conversations. The installation engineer just needs to start the DNN fitting routine.

This gets to the last bit of GPML that we highlight in Figure 1, the tools that facilitate model fitting on massive datasets: Out-of-sample (*OOS*) validation for model tuning, Stochastic Gradient Descent (*SGD*) for parameter optimization, and Graphical Processing Units (*GPUs*) and other computer hardware for massively parallel optimization. Each of these pieces is essential for the success of large-scale GPML. Although they are commonly associated with Deep Learning and DNNs (especially SGD and GPUs), these tools have developed in the context of many different ML algorithms. The rise of DNNs over alternative ML modeling schemes is partly due to the fact that, through trial and error, ML researchers have discovered that Neural Network models are especially well suited to engineering within the context of these available tools.[26]

OOS validation is a basic idea: you choose the best model specification by comparing predictions from models estimated on data that was not used during the model 'training' (fitting). This can be formalized as a cross-validation routine: you split the data into K 'folds', and then K times fit the model on all data but the K^{th} fold and evaluate its predictive performance (e.g., mean squared error or misclassification rate) on the left-out fold. The model with optimal average OOS performance (e.g., minimum error rate) is then deployed in practice.

ML's wholesale adoption of OOS validation as the arbitrator of model quality has freed the ML engineer from the need to *theorize* about model quality. Of course, this can create frustration and delays when you have nothing other than 'guess-and-test' as a method for model selection. But, increasingly, the requisite model search is not being executed by humans: it is done by additional ML routines. This either happens explicitly, in *AutoML*[9] frameworks that use simple auxiliary ML to predict OOS performance of the more complex target model, or implicitly by adding flexibility to the target model (e.g., making the tuning parameters part of the optimization objective). The fact that OOS validation provides a clear target to optimize against – a target which, unlike the in-sample likelihood, does not incentive over-fit – facilitates automated model tuning. It removes humans from the process of adapting models to specific datasets.

SGD optimization will be less familiar to most readers, but it is a crucial part of GPML. This class of algorithms allows models to be fit to data that is only observed in small chunks: you can train the model on a *stream* of data and avoid having to do *batch* computations on the entire dataset. This lets you estimate complex models on massive datasets. For subtle reasons, the engineering of SGD algorithms also tends to encourage robust and generalizable model fits (i.e., use of SGD discourages over-fit). We cover these algorithms in detail in a dedicated section below.

Finally, the GPUs: specialized computer processors have made massive-scale ML a reality and

continued hardware innovation will help push AI to new domains. Deep Neural Network training with Stochastic Gradient Descent involves massively *parallel* computations: many basic operations executed simultaneously across parameters of the network. Graphical Processing Units were devised for calculations of this type, in the context of video and computer graphics display where all pixels of an image need to be rendered simultaneously, in parallel. Although DNN training was originally a side use-case for GPUs (i.e., as an aside from their main computer graphics mandate), AI applications are now of primary importance for GPU manufacturers. Nvidia, for example, is a GPU company whose rise in market value has been driven by the rise of AI.

The technology here is not standing still. GPUs are getting faster and cheaper every day. We are also seeing the deployment of new chips that have been designed from scratch for ML optimization. For example, Field-Programmable Gate Arrays (FPGAs) are being used by Microsoft and Amazon in their data centers. These chips allow precision requirements to be set dynamically, thus efficiently allocating resources to high-precision operations and saving compute effort where you only need a few decimal points (e.g., in early optimization updates to the DNN parameters). As another example, Google's Tensor Processing Units (TPUs) are specifically designed for algebra with 'tensors', a mathematical object that occurs commonly in ML.[3]

One of the hallmarks of a general purpose technology is that it leads to broad industrial changes, both above and below where that technology lives in the supply chain. This is what we are observing with the new general purpose ML. Below, we see that chip makers are changing the type of hardware they create to suit these DNN-based AI systems. Above, GPML has led to a new class of ML-driven AI products. As we seek more real-world AI capabilities – self-driving cars, conversational business agents, intelligent economic marketplaces – domain experts in these areas will need to find ways to resolve their complex questions into structures of ML tasks. This is a role that economists and business professionals should embrace, where the increasingly user-friendly GPML routines become basic tools of their trade.

4 Deep Learning

We've stated that Deep Neural Networks are a key tool in GPML, but what exactly are they? And what makes them *deep*? In this section we will give a high level overview of these models. This is not a user guide. For that, we recommend the excellent recent textbook[11] by Goodfellow, Bengio, and Courville. This is a rapidly evolving area of research, and new types of Neural Network models and estimation algorithms are being developed at a steady clip. The excitement in this area, and considerable media and business hype, makes it difficult to keep track. Moreover, the tendency of ML companies and academics to proclaim every incremental change as 'completely brand new' has led to a messy literature that is tough for newcomers to navigate. But there is a general structure to Deep Learning, and a hype-free understanding of this structure should give you insight into the reasons for its success.

Neural Networks are simple models. Indeed, their simplicity is a strength: basic patterns facilitate fast training and computation. The model has linear combinations of inputs that are passed through non-linear activation functions called nodes (or, in reference to the human brain, neurons). A set of nodes taking different weighted sums of the same inputs is called a 'layer',

[3] A tensor is a multi-dimensional extension of a matrix – that is, a matrix is another name for a two-dimensional tensor.

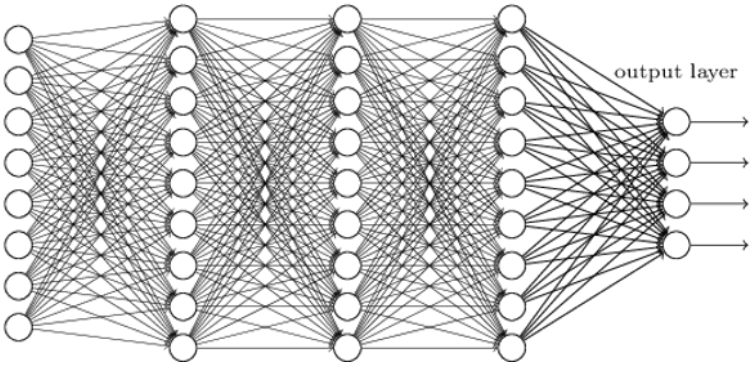

Figure 3: A five layer network, adapted from Nielsen [31].

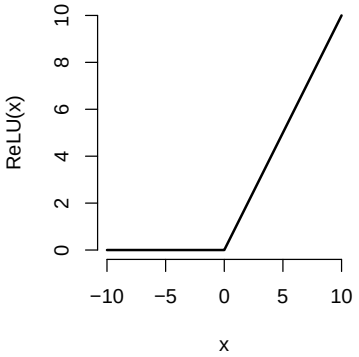

Figure 4: The ReLU function.

and the output of one layer's nodes becomes input to the next layer. This structure is illustrated in Figure 3. Each circle here is a node. Those in the input (furthest left) layer typically have a special structure; they are either raw data or data that has been processed through an additional set of layers (e.g., convolutions as we'll describe below). The output layer gives your predictions. In a simple regression setting, this output could just be $\hat{y}$, the predicted value for some random variable y, but DNNs can be used to predict all sorts of high-dimensional objects. As it is for nodes in input layers, output nodes also tend to take application-specific forms.

Nodes in the interior of the network have a 'classical' Neural Network structure. Say that $\eta_{hk}(\cdot)$ is the k^{th} node in interior layer h. This node takes as input a weighted combination of the output of the nodes in the previous layer of the network, layer $h-1$, and applies a *nonlinear* transformation to yield the output. For example, the ReLU (for 'rectified linear unit') node is by far the most common functional form used today; it simply outputs the maximum of it's input and zero, as shown in Figure 4.[4] Say z_{ij}^{h-1} is output of node j in layer $h-1$ for observation i. Then the corresponding output for the k^{th} node in the h^{th} layer can be written

$$z_{ik}^{h} = \eta_{hk}(\boldsymbol{\omega}_h' \boldsymbol{z}_i^{h-1}) = \max\left(0,\ \sum_j \omega_{hj} z_{ij}^{h-1}\right) \tag{1}$$

where ω_{hj} are the network *weights*. For a given network architecture – the structure of nodes and layers – these weights are the parameters that are updated during network training.

Neural Networks have a long history. Work on these types of models dates back to the mid 20th century, e.g., including Rosenblatt's Perceptron.[33] This early work was focused on networks as models that could mimic the actual structure of the human brain. In the late 1980s, advances in algorithms for *training* Neural Networks[34] opened the potential for these models to act as general pattern recognition tools rather than as a toy model of the brain. This led to a boom in Neural Network research, and methods developed during the 1990s are at the foundation of much of Deep Learning today.[18, 26] However, this boom ended in bust. Due to the gap between promised and realized results (and enduring difficulties in training networks on massive datasets) from the late 1990s Neural Networks became just one ML method among many. In applications

[4] In the 1990s, people spent much effort choosing amongst different node transformation functions. More recently, the consensus is that you can just use a simple and computationally convenient transformation (like ReLU). If you have enough nodes and layers the specific transformation doesn't really matter, so long as it is non-linear.

they were supplanted by more robust tools such as Random Forests, high-dimensional regularized regression, and a variety of Bayesian stochastic process models.

In the 1990s, one tended to add network complexity by adding *width*. A couple of layers (e.g., a single hidden layer was common) with a large number of nodes in each layer were used to approximate complex functions. Researchers had established that such 'wide' learning could approximate arbitrary functions[19] if you were able to train on enough data. The problem, however, was that this turns out to be an inefficient way to learn from data. The wide networks are very *flexible*, but they need a ton of data to tame this flexibility. In this way, the wide nets resemble traditional *nonparametric* statistical models like series and kernel estimators. Indeed, near the end of the 1990s, Radford Neal showed that certain Neural Networks converge towards Gaussian Processes, a classical statistical regression model, as the number of nodes in a single layer grows towards infinity.[30] It seemed reasonable to conclude that Neural Networks were just clunky versions of more transparent statistical models.

What changed? A bunch of things. Two non-methodological events are of primary importance: we got much more data (Big Data) and computing hardware became much more efficient (GPUs). But there was also a crucial methodological development: networks went *deep*. This breakthrough is often credited to 2006 work by Geoff Hinton and coauthors[17] on a network architecture that stacked many *pre-trained* layers together for a handwriting recognition task. In this pre-training, interior layers of the network are fit using an *unsupervised* learning task (i.e., dimension reduction of the inputs) before being used as part of the supervised learning machinery. The idea is analogous to that of Principal Components Regression: you first fit a low dimensional representation of $\boldsymbol{x}$, then use that low-D representation to predict some associated y. Hinton's scheme allowed researchers to train deeper networks than was previously possible.

This specific type of unsupervised pre-training is no longer viewed as central to deep learning. However, Hinton's paper opened many people's eyes to the potential for Deep Neural Networks: models with many layers, each of which may have different structure and play a very different role in the overall machinery. That is, a demonstration that one *could* train deep networks soon turned into a realization that one *should* add depth to models. In the following years, research groups began to show empirically and theoretically that depth was important for learning efficiently from data.[4] The *modularity* of a deep network is key: each layer of functional structure plays a specific role, and you can swap out layers like Lego blocks when moving across data applications. This allows for fast application-specific model development, and also for *transfer learning* across models: an internal layer from a network that has been trained for one type of image recognition problem can be used to hot-start a new network for a different computer vision task.

Deep Learning came into the ML mainstream with a 2012 paper by Krizhevsky, Sutskever, and Hinton[23] that showed their DNN was able to smash current performance benchmarks in the well-known ImageNet computer vision contest. Since then, the race has been on. For example, image classification performance has surpassed human abilities[15] and DNNs are now able to both recognize images and generate appropriate captions.[20]

The models behind these computer vision advances all make use of a specific type of *convolution* transformation. The raw image data (pixels) goes through multiple convolution layers before the output of those convolutions is fed into the more classical Neural Network architecture of (1) and Figure 3. A basic image convolution operation is shown in Figure 5: you use a *kernel* of weights to combine image pixels in a local area into a single output pixel in a (usually) lower-dimensional output image. So-called Convolutional Neural Networks[25] (CNNs) illustrate the

$$\begin{array}{|ccc|} \hline A & B & C \\ D & E & F \\ G & H & I \\ \hline \end{array} \star \begin{array}{|cc|} \hline \omega_1 & \omega_2 \\ \omega_3 & \omega_4 \\ \hline \end{array} = \begin{array}{|cc|} \hline \omega_1 A + \omega_2 B + \omega_3 D + \omega_4 E & \omega_1 B + \omega_2 C + \omega_3 E + \omega_4 F \\ \omega_1 D + \omega_2 E + \omega_3 G + \omega_4 H & \omega_1 E + \omega_2 F + \omega_3 H + \omega_4 I \\ \hline \end{array}$$

Figure 5: A basic convolution operation. The pixels A, B, etc, are multiplied and summed across kernel weights ω_k. The kernel here is applied to every 2×2 sub-matrix of our 'image'.

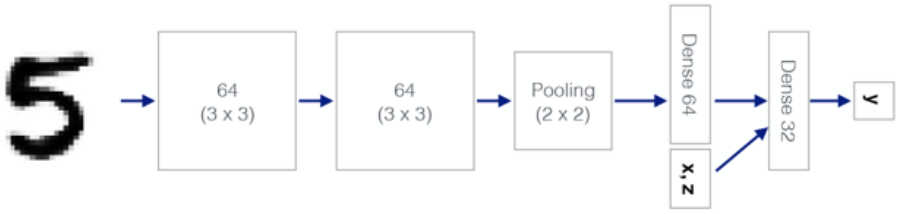

Figure 6: The network architecture used in Hartford et al. [13]. Variables $\boldsymbol{x}, \boldsymbol{z}$ contain structured business information (e.g., product IDs and prices) that is mixed with images of handwritten digits in our network.

strategy that makes Deep Learning so successful: it is convenient to stack layers of different specializations, such that image-specific functions (convolutions) can feed into layers that are good at representing generic functional forms. In a contemporary CNN, typically you will have multiple layers of convolutions feeding into ReLU activations and, eventually, into a *max pooling* layer constructed of nodes that output the maximum of each input matrix.[5] For example, Figure 6 shows the very simple architecture that we used in Hartford et al. [13] for a task that mixed digit recognition with (simulated) business data.

This is a theme of Deep Learning: the models use early-layer transformations that are specific to the input data format. For images, you use CNNs. For text data, you need to *embed* words into a vector space. This can happen through a simple word2vec transformation[28] (a linear decomposition on the matrix of co-occurrence counts for words, e.g., within three words of each other) or through a LSTM (Long-Short Term Memory) architecture[18] – models for sequences of words or letters that essentially mix a Hidden Markov Model (long) with an autoregressive process (short). And there are many other variants, with new architectures being developed every day.[6]

One thing should be clear: there is a lot of *structure* in DNNs. These models are *not* similar to the sorts of nonparametric regression models used by statisticians, econometricians, and in earlier ML. They are *semi-parametric*. Consider the cartoon DNN in Figure 7. The early stages in the network provide dramatic, and often linear, dimension reduction. These early stages are highly parametric: it makes no sense to take a convolution model for image data and apply it to, say, consumer transaction data. The output of these early layers is then processed through a series of classical Neural Network nodes, as in (1). These later network layers work like a traditional nonparametric regression: they expand the output of early layers to approximate arbitrary functional forms in the response of interest. Thus, the DNNs combine restrictive dimension reduction with

[5]CNNs are a huge and very interesting area. The textbook by Goodfellow et al. [11] is a good place to start if you want to learn more.

[6]For example, the new *Capsule* networks of Sabour et al. [35] replace the max-pooling of CNNs with more structured summarization functions.

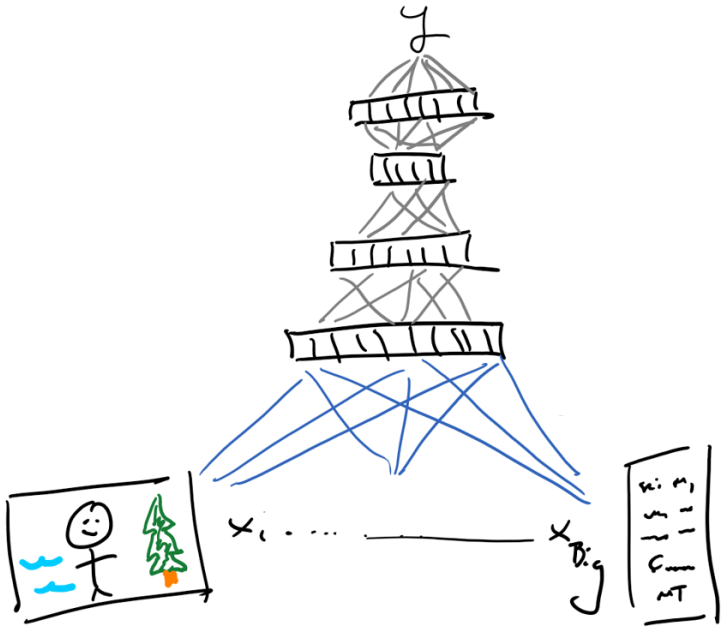

Figure 7: A cartoon of a DNN, taking as input images, structured data $x_1 \ldots x_{\text{big}}$, and raw document text.

flexible function approximation. The key is that both components are learned jointly.

As warned at the outset, we've covered only a tiny part of the area of Deep Learning. There is a ton of exciting new material coming out of both industry and academia. For a glimpse of what is happening in the field, browse the latest proceedings of NIPS (Neural Information Processing Systems, the premier ML conference) at `https://papers.nips.cc/`. You'll see quickly the massive breadth of current research. One currently hot topic is on uncertainty quantification for Deep Neural Networks, another is on understanding how imbalance in training data leads potentially biased predictions. Topics of this type are gaining prominence as DNNs are moving away from academic competitions and into real-world applications. As the field grows, and DNN model construction moves from a scientific to an engineering discipline, we'll see more need for this type of research that tells us when and how much we can trust the DNNs.

5 Stochastic Gradient Descent

To give a complete view of Deep Learning we need to describe the one algorithm that is relied upon for training all of the models: Stochastic Gradient Descent. SGD optimization is a twist on Gradient Descent (GD), the previously dominant method for minimizing any function that you can differentiate. Given a minimization objective $\mathcal{L}(\Omega)$, where Ω is the full set of model parameters, each iteration of a gradient descent routine updates from current parameters Ω_t as

$$\Omega_{t+1} = \Omega_t - C_t \nabla \mathcal{L}\big|_{\Omega_t} \tag{2}$$

where $\nabla\mathcal{L}\big|_{\Omega_t}$ is the gradient of $\mathcal{L}$ evaluated at the current parameters and C_t is a projection matrix that determines the size of the steps taken in the direction implied by $\nabla\mathcal{L}$.[7] We have the subscript t on C_t because this projection can be allowed to update during the optimization. For example, Newton's algorithm uses C_t equal to the matrix of objective second derivatives, $\nabla^2\mathcal{L}\big|_{\Omega_t}$.

It is often stated that Neural Networks are trained through 'back-propagation', which is not quite correct. Rather, they are trained through variants of Gradient Descent. Back-propagation[34], or back-prop for short, is a method for calculating gradients on the parameters of a network. In particular, back-prop is just an algorithmic implementation of your chain rule from calculus. In the context of our simple neuron from (1), the gradient calculation for a single weight ω_{hj} is

$$\frac{\partial\mathcal{L}}{\partial\omega_{hj}} \;=\; \sum_{i=1}^{n}\frac{\partial\mathcal{L}}{\partial z_{ij}^{h}}\frac{\partial z_{ij}^{h}}{\partial\omega_{hj}} \;=\; \sum_{i=1}^{n}\frac{\partial\mathcal{L}}{\partial z_{ij}^{h}} z_{ij}^{h-1}\mathbb{1}_{\left[0<\sum_j \omega_{hj} z_{ij}^{h-1}\right]}. \tag{3}$$

Another application of the chain rule can be used to expand $\partial\mathcal{L}/\partial z_{ij}^{h}$ as $\partial\mathcal{L}/\partial z_{ij}^{h+1} * \partial z_{ij}^{h+1}/\partial z_{ij}^{h}$, and so on until you have written the full gradient as a product of layer-specific operations. The directed structure of the network lets you efficiently calculate all of the gradients by working backwards layer by layer, from the response down to the inputs. This recursive application of the chain rule, and the associated computation recipes, make up the general back-prop algorithm.

In statistical estimation and ML model-training, $\mathcal{L}$ typically involves a loss function that *sums* across data observations. For example, assuming an ℓ_2 (ridge) regularization penalty on the parameters, the *minimization* objective corresponding to regularized likelihood maximization over n independent observations d_i (e.g., $d_i = [\boldsymbol{x}_i, y_i]$ for regression) can be written as

$$\mathcal{L}(\Omega) \equiv \mathcal{L}\left(\Omega; \{d_i\}_{i=1}^{n}\right) = \sum_{i=1}^{n}\left[-\log \mathrm{p}(z_i|\Omega) + \lambda\|\Omega\|_2^2\right] \tag{4}$$

where $\|\Omega\|_2^2$ is the sum of all squared parameters in Ω. More generally, $\mathcal{L}\left(\Omega; \{d_i\}_{i=1}^{n}\right)$ can consist of any loss function that involves summation over observations. For example, to model predictive uncertainty we often work with quantile loss. Define $\tau_q(\boldsymbol{x};\Omega)$ as the *quantile function*, parametrized by Ω, that maps from covariates $\boldsymbol{x}$ to the q^{th} quantile of the response y,

$$\mathrm{P}(y < \tau_q(\boldsymbol{x};\Omega) \mid \boldsymbol{x}) = q. \tag{5}$$

We fit τ_q to minimize the regularized quantile loss function (again assuming a ridge penalty),

$$\mathcal{L}\left(\Omega; \{d_i\}_{i=1}^{n}\right) = \sum_{i=1}^{n}\left[\left(y_i - \tau_q(\boldsymbol{x}_i;\Omega)\right)\left(q - \mathbb{1}_{[y_i<\tau_q(\boldsymbol{x}_i;\Omega)]}\right) + \lambda\|\Omega\|_2^2\right]. \tag{6}$$

The very common 'sum of squared errors' criterion, possibly regularized, is another loss function that fits this pattern of summation over observations.

In all of these cases, the gradient calculations required for the updates in (2) involve sums over all n observations. That is, each calculation of $\nabla\mathcal{L}$ requires an order of n calculations. For example,

[7] If $\Omega = [\omega_1 \cdots \omega_p]$, then $\nabla\mathcal{L}(\Omega) = \left[\frac{\partial L}{\partial\omega_1} \cdots \frac{\partial L}{\partial\omega_p}\right]$. The *Hessian* matrix, $\nabla^2\mathcal{L}$, has elements $[\nabla^2\mathcal{L}]_{jk} = \frac{\partial L^2}{\partial\omega_j\partial\omega_k}$.

in a ridge penalized linear regression where $\Omega = \boldsymbol{\beta}$, the vector of regression coefficients, the j^{th} gradient component is,

$$\frac{\partial \mathcal{L}}{\partial \beta_j} = \sum_{i=1}^{n} \left[(y_i - \boldsymbol{x}_i' \boldsymbol{\beta}) x_j + \lambda \beta_j \right]. \tag{7}$$

The problem for massive datasets is that when n is really big these calculations become prohibitively expense. The issue is aggravated when, as it is for DNNs, Ω is high dimensional and there are complex calculations required in each gradient summand. GD is the best optimization tool that we've got, but it becomes computationally infeasible for massive datasets.

The solution is to replace the actual gradients in (2) with *estimates* of those gradients based upon a subset of the data. This is the SGD algorithm. It has a long history, dating back to the Robbins-Munro[32] algorithm proposed by a couple of statisticians in 1951. In the most common versions of SGD, the full-sample gradient is simply replaced by the gradient on a smaller sub-sample. Instead of calculating gradients on the full-sample loss, $\mathcal{L}(\Omega; \{d_i\}_{i=1}^n)$, we descend according to sub-sample calculations:

$$\Omega_{t+1} = \Omega_t - C_t \nabla \mathcal{L}\left(\Omega; \{d_{i_b}\}_{b=1}^B\right)\Big|_{\Omega_t} \tag{8}$$

where $\{d_{i_b}\}_{b=1}^B$ is a *mini-batch* of observations with $B \ll n$. The key mathematical result behind SGD is that, so long as the sequence of C_t matrices satisfy some basic requirements, the SGD algorithm will converge to a local optimum whenever $\nabla \mathcal{L}\left(\Omega; \{d_{i_b}\}_{b=1}^B\right)$ is an *unbiased* estimate of the full sample gradient.[8] That is, SGD convergence relies upon

$$\mathbb{E}\left[\frac{1}{B}\nabla \mathcal{L}\left(\Omega; \{d_{i_b}\}_{b=1}^B\right)\right] = \mathbb{E}\left[\frac{1}{n}\nabla \mathcal{L}\left(\Omega; \{d_i\}_{i=1}^n\right)\right] = \mathbb{E}\nabla \mathcal{L}\left(\Omega; d\right) \tag{9}$$

where the last term here refers to the *population* expected gradient – that is, the average gradient for observation d drawn from the true Data Generating Process.

To understand why SGD is so preferable to GD for Machine Learning, it helps to discuss how Computer Scientists think about the *constraints* on estimation. Statisticians and Economists tend to view sample size (i.e., lack of data) as the binding constraint on their estimators. In contrast, in many ML applications the data is practically unlimited and continues to grow during system deployment. Despite this abundance, there is a fixed computational budget (or the need to update in near-real-time for streaming data), such that we can only execute a limited number of operations when crunching through the data. Thus, in ML, the binding constraint is the amount of computation rather than the amount of data.

SGD trades faster updates for a slower per-update convergence rate. As nicely explained in a 2008 paper by Bousquet and Boutteau[5], this trade is worthwhile when the faster updates allow you to expose your model to more data than would otherwise be possible. To see this, note that the mini-batch gradient $B^{-1}\nabla \mathcal{L}\left(\Omega; \{d_{i_b}\}_{b=1}^B\right)$ has a much higher variance than the full-sample gradient, $n^{-1}\nabla \mathcal{L}\left(\Omega; \{d_i\}_{i=1}^n\right)$. This variance introduces noise into the optimization updates. As a result, for a fixed data sample n, the GD algorithm will tend to take far fewer iterations than SGD to get to a minimum of the *in-sample* loss, $\mathcal{L}\left(\Omega; \{d_i\}_{i=1}^n\right)$. However, in DNN training we don't really care about the in-sample loss. We really want to minimize future prediction loss – that is,

[8] You can actually get away with biased gradients. In Hartford et al. [13] we find that trading bias for variance can actually improve performance. But this is tricky business and in any case the bias must be kept very small.

we want to minimize the *population* loss function $\mathbb{E}\mathcal{L}(\Omega; d)$. And the best way to understand the population loss is to see as much data as possible. Thus if the variance of the SGD updates is not too large, it is more valuable to spend computational effort streaming through more data than to spend it on minimizing the variance of each individual optimization update.

This is related to an important high-level point about SGD: the nature of the algorithm is such that engineering steps taken to improve *optimization* performance will tend to also improve *estimation* performance. The same tweaks and tricks that lower the variance of each SGD update will lead to fitted models that generalize better when predicting new unseen data. The 'train faster, generalize better' paper by Hardt, Recht, and Singer[12] explains this phenomenon within the framework of algorithm stability. For SGD to converge in fewer iterations means that the gradients on new observations (new mini-batches) are approaching zero more quickly. That is, faster SGD convergence means by definition that your model fits are generalizing better to unseen data. Contrast this with full-sample GD, e.g., for likelihood maximization: faster convergence implies only quicker fitting on your current sample, potentially over-fitting for future data. A reliance on SGD has made it relatively easy for Deep Learning to progress from a Scientific to Engineering discipline. Faster is better, so the engineers tuning SGD algorithms for DNNs can just focus on convergence speed.

On the topic of tuning SGD: real-world performance is very sensitive to the choice of C_t, the projection matrix in (8). For computational reasons, this matrix is usually diagonal (i.e., it has zeros off of the diagonal) such that entries of C_t dictate your *step-size* in the direction of each parameter gradient. SGD algorithms have often been studied theoretically under a single step-size, such that $C_t = \gamma_t I$ where γ_t is a scalar and I is the identity matrix. Unfortunately, this simple specification will under-perform and even fail to converge if γ_t is not going towards zero at a precise rate.[42] Instead, practitioners make use of algorithms where $C_t = [\gamma_{1t} \cdots \gamma_{pt}]I$, with p the dimension of Ω, and each γ_{jt} is chosen to approximate $\partial^2 \mathcal{L} / \partial \omega_j^2$, the corresponding diagonal element of the Hessian matrix of loss-function second derivatives (i.e., what would be used in a Newton's algorithm). The ADAGRAD paper[8] provides a theoretical foundation for this approach and suggests an algorithm for specifying γ_{jt}. Most Deep Learning systems make use of ADAGRAD-inspired algorithms, such as ADAM[22], that combine the original algorithm with heuristics that have been shown empirically to improve performance.

Finally, there is another key trick to DNN training: *Dropout*. This procedure, proposed by researchers[39] in Hinton's lab at the University of Toronto, involves introduction of random noise into each gradient calculation. For example, 'Bernoulli dropout' replaces current estimates ω_{tj} with $w_{tj} = \omega_{tj} * \xi_{tj}$ where ξ_{tj} is a Bernoulli random variable with $\mathrm{p}(\xi_{tj} = 1) = c$. Each SGD update from (8) then uses these parameter values when evaluating the gradient, such that

$$\Omega_{t+1} = \Omega_t - C_t \nabla f\left(\Omega; \{d_{i_b}\}_{b=1}^{B}\right)\Big|_{W_t}, \tag{10}$$

where W_t is the noised-up version of Ω_t, with elements w_{tj}.

Dropout is used because it has been observed to yield model fits that have lower out-of-sample error rates (so long as you tune c appropriately). Why does this happen? Informally, Dropout acts as a type of implicit regularization. An example of explicit regularization is parameter penalization: to avoid over-fit, the minimization objective for DNNs almost always has a $\lambda \|\Omega\|_2^2$ ridge penalty term added to the data-likelihood loss function. Dropout plays a similar role. By forcing SGD updates to ignore a random sample of the parameters, it prevents over-fit on any individual

parameter.[9] More rigorously, it has recently been established by a number of authors[21] that SGD with dropout corresponds to a type of 'variational Bayesian Inference'. That means that dropout SGD is solving to find the posterior *distribution* over Ω rather than a point estimate.[10] As interest grows around uncertainty quantification for DNNs, this interpretation of Dropout is one option for bringing Bayesian inference into Deep Learning.

6 Reinforcement Learning

As our final section on the elements of Deep Learning, we will consider how these AI systems generate their own training data through a mix of experimentation and optimization. Reinforcement Learning (RL) is the common term for this aspect of AI. RL is sometimes used to denote specific algorithms, but we are using it to refer to the full area of active data collection.

The general problem can be formulated as an reward maximization task. You have some policy or 'action' function, $d(x_t;\Omega)$, that dictates how the system responds to 'event' t with characteristics x_t. The event could be a customer arriving on your website at a specific time, or a scenario in a video game, etc. After the event, you observe 'response' y_t and the reward is calculated as $r(d(x_t,;\Omega),y_t)$. During this process you are accumulating data and *learning* the parameters Ω, so we can write Ω_t as the parameters used at event t. The goal is that this learning converges to some optimal reward-maximizing parametrization, say $\Omega^\star$, and that this happens after some T events where T is not too big – i.e., so that you minimize *regret*,

$$\sum_{t=1}^{T}\left[r\left(d(x_t;\Omega^\star),y_t\right)-r\left(d(x_t;\Omega_t),y_t\right)\right]. \tag{11}$$

This is a very general formulation. We can map it to some familiar scenarios. For example, suppose that the event t is a user landing on your website. You would like to show a banner advertisement on the landing page, and you want to show the ad that has the highest probability of getting clicked by the user. Suppose that there are J different possible ads you can show, such that your action $d_t = d(x_t;\Omega_t) \in \{1,\ldots,J\}$ is the one chosen for display. The final reward is $y_t = 1$ if the user clicks the ad and $y_t = 0$ otherwise.[11]

This specific scenario is a *Multi-Armed Bandit* (MAB) set-up, so-named by analogy to a casino with many slot machines of different payout probabilities (the casino is the bandit). In the classic MAB (or simply 'bandit') problem, there are no covariates associated with each ad and each user, such that you are attempting to optimize towards a single ad that has highest click probability across all users. That is, ω_j is $\mathrm{p}(y_t = 1 \mid d_t = j)$, the generic click probability for ad j, and you want to set d_t to the ad with highest ω_j. There are many different algorithms for bandit optimization. They use different heuristics to balance *exploitation* with *exploration*. A fully exploitive algorithm is greedy: it always takes the currently estimated best option without any consideration of uncertainty. In our

[9] This seems to contradict our earlier discussion about minimizing the variance of gradient estimates. The distinction is that we want to minimize variance due to noise in the data, but here we are introducing noise in the parameters *independent* of the data.

[10] It is a strange variational distribution, but basically the posterior distribution over Ω becomes that implied by W, with elements ω_j multiplied by random Bernoulli noise.

[11] This application, on the news website `MSN.com` with headlines rather than ads, motivates much of the RL work in Agarwal et al. [1].

simple advertising example, this implies always converging to the first ad that ever gets clicked on. A fully exploratory algorithm always randomizes the ads and it will never converge to a single optimum. The trick to bandit learning is finding a way to balance between these two extremes.

A classic bandit algorithm, and one which gives solid intuition into RL in general, is Thompson Sampling.[41] Like many tools in RL, Thompson Sampling uses Bayesian inference to model the accumulation of knowledge over time. The basic idea is simple: at any point in the optimization process you have a probability distribution over the vector of click rates, $\boldsymbol{\omega} = [\omega_1 \dots \omega_J]$, and you want to show each ad j in proportion to the probability that ω_j is the largest click rate. That is, with $y^t = \{y_s\}_{s=1}^t$ denoting observed responses at time t, you want to have

$$\mathrm{p}(d_{t+1} = j) \propto \mathrm{p}\left(\omega_j = \max\{\omega_k\}_{k=1}^J \mid y^t\right), \tag{12}$$

such that an ad's selection probability is equal to the posterior probability that it is the best choice. Since the probability in (12) is tough to calculate in practice (the probability of a maximum is not an easy object to analyze), Thompson Sampling uses Monte Carlo estimation. In particular, you draw a sample of ad-click probabilities from the posterior distribution at time t,

$$\boldsymbol{\omega}_{t+1} \sim \mathrm{p}(\boldsymbol{\omega} \mid y^t), \tag{13}$$

and set $d_{t+1} = \mathrm{argmax}_j\ \omega_{t+1\,j}$. For example, suppose that you have a $\mathrm{Beta}(1,1)$ prior on each ad's click rate (i.e., a uniform distribution between zero and one). At time t, the posterior distribution for the j^{th} ad's click rate is

$$\mathrm{P}(\omega_j | d^t, y^t) = \mathrm{Beta}\left(1 + \sum_{s=1}^t \mathbb{1}_{[d_s=j]} y_s,\ 1 + \sum_{s=1}^t \mathbb{1}_{[d_s=j]}(1-y_s)\right). \tag{14}$$

A Thompson sampling algorithm draws $\omega_{t+1\,j}$ from (14) for each j and then shows the ad with highest sampled click rate.

Why does this work? Think about scenarios where an ad j would be shown at time t – i.e., when the sampled $\omega_{t\,j}$ is largest. This can occur if there is a lot of uncertainty about ω_j, in which case high probabilities have non-trivial posterior weight, or if the expected value of ω_j is high. Thus Thompson Sampling will naturally balance between exploration and exploitation. There are many other algorithms for obtaining this balance. For example, Agarwal et al. [1] survey methods that work well in the *contextual* bandit setting where you have covariates attached to events (such that action-payoff probabilities are event-specific). The options considered include ε-greedy search, which finds a predicted optimal choice and explores within a neighborhood of that optimum, and a Bootstrap based algorithm that is effectively a nonparametric version of Thompson Sampling.

Another large literature looks at so-called Bayesian Optimization.[40] In these algorithms, you have an unknown function $r(x)$ that you'd like to maximize. This function is modeled using some type of flexible Bayesian regression model, e.g., a Gaussian Process. As you accumulate data, you have a posterior over the 'response surface' r at all potential input locations. Suppose that, after t function realizations, you have observed a maximal value $r_{\max}$. This is your current best option, but you want to continue exploring to see if you can find a higher maximum. The Bayesian Optimization update is based on the *Expected Improvement* statistic,

$$\mathbb{E}[\max(0, r(x) - r_{\max})], \tag{15}$$

the posterior expectation of improvement at new location x, thresholded below at *zero*. The algorithm evaluates (15) over a grid of potential x locations, and you choose to evaluate $r(x_{t+1})$ at the location x_{t+1} with highest Expected Improvement. Again, this balances exploitation with exploration: the statistic in (15) can be high if $r(x)$ has high variance or a high mean (or both).

These RL algorithms are all described in the language of optimization, but it is possible to map many learning tasks to optimization problems. For example, the term *Active Learning* is usually used to refer to algorithms that choose data to minimize some estimation variance (e.g., the average prediction error for a regression function over a fixed input distribution). Say $f(x;\Omega)$ is your regression function, attempting to predict response y. Then your *action* function is simply prediction, $d(x;\Omega) = f(x;\Omega)$, and your optimization goal could be to minimize the squared error – i.e., to maximize $r(d(x;\Omega),y) = -(y - f(x;\Omega))^2$. In this way, active learning problems are special cases of the RL framework.

From a business and economic perspective, RL is interesting (beyond its obvious usefulness) for assigning a *value* to new data points. In many settings the rewards can be mapped to actual monetary value: e.g., in our advertising example where the website receives revenue-per-click. RL algorithms assign a dollar value to data observations. There is a growing literature on markets for data, e.g., including the 'data-is-labor' proposal in Lanier [24]. It seems useful for future study in this area to take account of how currently deployed AI systems assign relative data value. As a high-level point, the valuation of data in RL depends upon the *action* options and potential *rewards* associated with these actions. The value of data is only defined in a specific context.

The bandit algorithms described above are vastly simplified in comparison to the type of RL that is deployed as part of a Deep Learning system. In practice, when using RL with complex flexible functions like DNNs you need to be very careful to avoid over exploitation and early convergence.[29] It is also impossible to do a comprehensive search through the super high-dimensional space of optional values for the Ω that parametrizes a DNN. However, approaches such as that in van Seijen et al. [43] and Silver et al. [38] show that if you impose *structure* on the full learning problem then it can be broken into a number of simple composite tasks, each of which is solvable with RL. As we discussed earlier, there is an undeniable advantage to having large fixed data assets that you can use to hot-start your AI (e.g., data from a search engine or social media platform). But the exploration and active data collection of RL is essential when tuning an AI system to be successful in specific contexts. These systems are taking actions and setting policy in an uncertain and dynamic world. As statisticians, scientists and economists are well aware, without constant experimentation it is not possible to learn and improve.

7 AI in context

This chapter has provided a primer on the key ingredients of AI. We have also been pushing some general points. First, the current wave of ML-driven AI should be viewed as a new class of products growing up around a new general purpose technology: large-scale, fast, and robust Machine Learning. AI is not Machine Learning, but general purpose ML, specifically Deep Learning, is the electric motor of AI. These ML tools are going to continue to get better, faster, and cheaper. Hardware and Big Data resources are adapting to the demands of DNNs, and self-service ML solutions are available on all of the major Cloud Computing platforms. Trained DNNs might become a commodity in the near term future, and the market for Deep Learning could get wrapped up in

the larger battle over market share in Cloud Computing services.

Second, we are still waiting for true end-to-end Business AI solutions that drive a real increase in productivity. AI's current 'wins' are mostly limited to settings with high amounts of explicit structure, like board and video games.[12] This is changing, as companies like Microsoft and Amazon produce semi-autonomous systems that can engage with real business problems. But there is still much work to be done, and the advances will be made by those who can impose structure on these complex business problems. That is, for business AI to succeed we need to combine the GPML and Big Data with people who know the rules of the 'game' in their business domain.

Finally, all of this will have significant implications for the role of economics in industry. In many cases, the economists are those who can provide structure and rules around messy business scenarios. For example, a good structural Econometrician[27, 16, 7] uses economic theory to break a substantiative question into a set of *measurable* (i.e., identified) equations with parameters that can be estimated from data. In many settings, this is *exactly* the type of work-flow required for AI. The difference is that, instead of being limited to basic linear regression, these measurable pieces of the system will DNNs that can actively experiment and generate their own training data. The next generation of economists needs to be comfortable in knowing how to apply economic theory to obtain such structure, and how to translate this structure into recipes that can be automated with ML and RL. Just as Big Data led to Data Science, a new discipline combining Statistics and Computer Science, AI will require interdisciplinary pioneers who can combine Economics, Statistics, and Machine Learning.

References

[1] Alekh Agarwal, Daniel Hsu, Satyen Kale, John Langford, Lihong Li, and Robert Schapire. Taming the monster: A fast and simple algorithm for contextual bandits. In *International Conference on Machine Learning*, pages 1638–1646, 2014.

[2] Susan Athey. Beyond prediction: Using big data for policy problems. *Science*, 355:483–485, 2017.

[3] Susan Athey and Guido Imbens. Recursive partitioning for heterogeneous causal effects. *Proceedings of the National Academy of Sciences*, 113:7353–7360, 2016.

[4] Yoshua Bengio, Yann LeCun, et al. Scaling learning algorithms towards ai. *Large-scale kernel machines*, 34(5):1–41, 2007.

[5] Olivier Bousquet and Léon Bottou. The tradeoffs of large scale learning. In *Advances in neural information processing systems*, pages 161–168, 2008.

[6] Timothy Bresnahan. General purpose technologies. *Handbook of the Economics of Innovation*, 2:761–791, 2010.

[7] Angus Deaton and John Muellbauer. An almost ideal demand system. *The American economic review*, 70(3):312–326, 1980.

[12] The exception to this is web search, which has been effectively solved through AI.

[8] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[9] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[10] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] Moritz Hardt, Ben Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *International Conference on Machine Learning*, pages 1225–1234, 2016.

[13] Jason Hartford, Greg Lewis, Kevin Leyton-Brown, and Matt Taddy. Deep iv: A flexible approach for counterfactual prediction. In *International Conference on Machine Learning*, pages 1414–1423, 2017.

[14] John Haugeland. *Artificial Intelligence: The Very Idea*. MIT Press, 1985.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[16] James J Heckman. Sample selection bias as a specification error (with an application to the estimation of labor supply functions), 1977.

[17] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

[19] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[20] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[21] Alex Kendall and Yarin Gal. What uncertainties do we need in bayesian deep learning for computer vision? *arXiv preprint arXiv:1703.04977*, 2017.

[22] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations (ICLR)*, 2015.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[24] Jaron Lanier. *Who Owns the Future*. Simon & Schuster, 2014.

[25] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[26] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[27] Daniel McFadden. Econometric models for probabilistic choice among products. *Journal of Business*, pages S13–S29, 1980.

[28] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[30] Radford M Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

[31] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[32] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[33] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65:386, 1958.

[34] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[35] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In *Advances in Neural Information Processing Systems*, pages 3857–3867, 2017.

[36] Steven L Scott. A modern bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26(6):639–658, 2010.

[37] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529: 484–489, 2016.

[38] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

[39] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[40] Matt Taddy, Herbert KH Lee, Genetha A Gray, and Joshua D Griffin. Bayesian guided pattern search for robust local optimization. *Technometrics*, 51(4):389–401, 2009.

[41] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25:285–294, 1933.

[42] Panagiotis Toulis, Edoardo Airoldi, and Jason Rennie. Statistical analysis of stochastic gradient methods for generalized linear models. In *International Conference on Machine Learning*, pages 667–675, 2014.

[43] Harm van Seijen, Mehdi Fatemi, Joshua Romoff, Romain Laroche, Tavian Barnes, and Jeffrey Tsang. Hybrid reward architecture for reinforcement learning. *arXiv:1706.04208*, 2017.