Table of Contents

Summary

Artificial Intelligence (AI) and Machine Learning (ML) technologies are key to many modern engineering projects due to their ability to solve many problems that are difficult or impossible with other methods. While most engineers will find themselves enjoying a significant overlap between these techniques and their existing skill set, they are also liable to find that AI and Machine Learning is its own field with its own unique demands and (often hidden) pitfalls. While there are many resources available for self teaching, it is generally assumed the practitioner is either an absolute beginner to engineering, or already a seasoned expert in AI and ML. In this document, we provide a practical guide to AI and Machine learning for electronic systems engineers who already have a strong base of knowledge in electronic systems but no specialised expertise. This guide will be practice focused, with the goal of helping engineers to make good decisions and avoid problems. The guide will cover, among many others areas:

- Technical Language and Core Concepts
- Data, Collection and Common Problems
- Deploying AI and Machine Learning, with Worked Examples
- Existing resources, where to find them and how best to use them
- Common Pitfalls, how to avoid and how to solve

Target Audience

The target audience for this guide is electronic systems engineers with little to no specific expertise in AI and Machine Learning techniques. It will be assumed, because of this background, that readers will have a base level of competence in programming and mathematics, though the guide will err on the side of caution in respect of assumed knowledge. We may, for example, assume our reader has knowledge of concepts in basic calculus, but are unlikely to assume that they are able to remember any specific formula. We are also assuming that the primary interest in practically deploying these techniques rather than understanding the theory and context of their development. For readers interested in a more theoretical treatment, we list several texts in the resources section appropriate to a range of different levels of background knowledge.

Introduction

The terms Artificial Intelligence and Machine Learning both lend themselves to the idea that we are creating a program that can, in some respect, think and learn in the same way a person can. While it may be possible to create systems that are capable of this, the current technologies that are deployed in practice are much more restricted in scope. Current, practical Artificial Intelligence and Machine Learning technology generally refers to programs that are of a statistical, or statistically adjacent nature, and that use data to fit some specified mathematical model. While these techniques are proving extremely valuable in solving otherwise difficult or intractable problems, the fundamentals of the techniques are neither exceptionally novel nor complicated. In this guide, we aim to show just how simple it can be to deploy these techniques in practice, especially for an audience that already has knowledge of electronic systems engineering.

It is important to acknowledge that, despite the above, no specific definition of Artificial Intelligence and Machine Learning that is universally agreed on exists. Furthermore, lacking any significant advancement in our understanding of what cognition, intelligence, and learning actually mean, this state of affairs is likely to continue. Given the practical focus of this guide, it would not be profitable for us to enter into this debate at any length. We will therefore, universally use the term “AI and Machine Learning” to refer to algorithms that are the subject of this guide, and that learn to **parameterize** a mathematical model from data.

This guide is structured into 5 sections. In the first section, we discuss the key concepts and terminology that are used in AI and Machine Learning and briefly discuss how they come together to motivate us to create the field in its current form. With this important background covered, we then move on to discussing the practicalities of AI and Machine Learning projects at a high level. We discuss where it’s appropriate to deploy these technologies, a high level workflow and decision making process, and what management of these projects looks like. Having covered this, we then begin to drill down into the details of these projects. We open this discussion with a review of the key role data, and how it can be managed. This is followed with an extensive discussion on the implementation and decision making process of deployment of these algorithms with worked examples. We close with a list of pitfalls, common problems and solutions, and a link to useful further resources.

Overview: Key Concepts and Terminology

In this section, we introduce and overview some of the key concepts and terminology that are important in understanding and discussing AI and machine learning technology. This section is not intended to serve as an exhaustive discussion of the topics in question, but as an introduction to the key points and vocabulary that will frame the rest of the discussion on this topic. Key terms and concepts will be **highlighted** for the reader, with links to a more detailed description of the term in the appendix.

To briefly review what we discussed in the introduction: the type of intelligence and learning in algorithms that we are interested in in this manual are algorithms that learn mathematical models of the world from some data, in order to make predictions about other unseen or future data. One important idea that we need to consider first is **structured data** and **unstructured data**.

TechNES AI Best Practices Group:

Breakaway: Structured vs Unstructured Data AI and Machine Learning models are no different from any other computer program in that they require their input data to follow a consistent format. Unfortunately, data collected in the real world rarely follows the type of structure and organization that is needed for ingestion by an AI or Machine learning algorithm, and in many cases the work done to transform data into an appropriate structured format is some of the most important work done in any AI and Machine Learning pipeline. We make this distinction between data that has been put into a useful structured format as **structured data**, and data that exists in a raw, unprocessed format as **unstructured data**.

When dealing with data in the real world, we will often split it up into categories or types. One such distinction often made that is especially important in the context of AI and Machine Learning is the split of data into **continuous** data and **discrete** data. Continuous data can take on any number of infinite values across a given range, for example, a measure of rainfall per hour. Discrete data on the other hand is any type of data that falls into a fixed number of categories. These categories can be both **ordinal** data in which there is a natural ordering between the categories (shoe size, for example), and **nominal** data, where the categories are distinct (eye color, for example). While this distinction is important for many parts of AI and Machine Learning, the distinction between whether an AI and Machine Learning algorithm is trying to predict continuous and discrete data is so important that it has its own nomenclature of **regression** and **classification** algorithms respectively.

Breakaway: Regression vs Classification Algorithms The distinction between **regression** (continuous output data) and **classification** (discrete output data) is particularly important in AI and Machine Learning algorithms, because the type of data that the algorithm outputs has a significant effect on how it must function. Notably, some algorithms (e.g. Support Vector Machines) are only designed to function in one of these modalities, and require significant adaptations to perform (likely very poorly) in the other.

While we have been discussing some of the concepts and terminology around data to this point, we have used the terms “learn”, “learning” and “learning from data” to describe what our algorithms do without really making it explicit what we actually mean by this. One of the reasons that we’ve avoided doing this is that “learning” in the context we’re discussing it is conveniently, without further qualifiers, a term that covers several different ideas. These differences stem from the way that we use data in order to “learn”. The most prominent of two of these ideas are **supervised learning** and **unsupervised learning**, which are concerned whether we learn from data that list the correct output the algorithms should produce for some given input data (**labeled data**), or simply the input data themselves (**unlabeled data**).

Breakaway: Supervised vs Unsupervised vs Reinforcement vs Other Learning We use the nomenclature of **Supervised** vs **Unsupervised** (vs others) to describe the way in which our algorithms are learning. In Supervised learning, we learn from matched input data/output data pairs, data for which we already have the correct output the algorithms should predict for a set of given inputs (“learning by example”). We call this data **labeled data**, because our set of input data is labeled with the corresponding correct solutions.For example, we might be interested in predicting the future prices of the stock market from economic indications, by looking at how these economic indicators have predicted its historical past prices. In Unsupervised learning, we only have access to the input data without any corresponding output solution attached. We call this data **unlabeled data**, and our unsupervised learning algorithms and are generally interested in predicting some quality of this data (“pattern learning”). For example, we might be detecting unusual anomalies of electrical usage in the grid.

While it is generally preferable to use supervised learning when we can because learning by example is easier, there are many situations in which unsupervised approaches are more appropriate. Even putting aside the fact that unlabeled data is easier to collect (since we don’t need to label it), for many problems supervised approaches are simply not practical. In our electrical grid example above, it would be infeasible to train a supervised model to do similar anomaly detection. By definition, anomalies are rare and unusual data points that fall outside of the usual observations in the data. Creating a labeled dataset of them would be both impractical, and any supervised algorithm that used it would be prescriptive - it would only catch anomalies similar to anomalies we’ve trained on, where an unsupervised approach instead catches ones that are dissimilar to everything we’ve seen so far.

There are also several other learning approaches that fit within the supervised/unsupervised dichotomy discussed so far. A common one is **Reinforcement Learning**. In Reinforcement Learning, the algorithm is not fed a set of data, but selects which piece of data it wants to learn from in future from the pieces of data it has had up until now. Another common paradigm is **semi-supervised learning**, in which an algorithm learns from some set data that is labeled, and some (usually larger) set of data that is unlabelled.

Putting it Together: Creating Modern AI and Machine Learning

WIP

Running AI Projects

Should I AI?

AI Project Decisions

AI Project Workflow

Data

The Importance of Data

Collecting Data

Managing Data

Building AI and Machine Learning Models

Decisions

Decision Making Flowchart

Neural Networks

Architectural Principles of Neural Networks

Here we break down some of the key ideas that go into building a more sophisticated neural network than the basic MLP we used in our first example. Understanding these will allow you to see how even the world's most complex and capable neural networks are put together.

- **Convolutional Blocks**
    - **What:** An architectural design shortcut - a combination of convolutional layers, pooling layers and regularisers packaged into a block which is repeated throughout the network.
    - **Why:** Convolutional layers are used to extract local patterns in the input data. By stacking many convolutional layers on top of each other, higher level relationships are able to be recognised. In practice, this means the network is able to perform better (more complex) pattern recognition, which is what machine learning is all about. Using a repeating block structure like the one pictured below has been found to be very effective. Each block contains multiple convolutions and a pooling layer to reduce dimensionality. Each element of the block is optional and customisable, but the principle of repeating blocks remains the same.

- **Skip Connections**
    - **What:** Passing the output of one layer in the network directly to later layers in the network, ‘skipping’ over the intermediate layers. This can be done through addition, which requires the dimensions of the layers to be equal, or through appending, which increases the dimension size. Appending is the safer choice, though comes at greater computational cost.
    - **Why:** As we make our networks deeper, we are able to extract higher level features. This is extremely powerful, but some new issues begin to emerge. Firstly, the low level information can get lost on the way. Secondly, we can run into the vanishing gradient problem. A neat solution to diminish both of these issues is to use skip connections.
- **Bottlenecks**
    - **What:** A bottleneck refers to the shape of the network (big to small). Often followed by more layers to build the network size back up (small to big). The bottleneck itself is the smallest layer, which can also be called the encoding layer.
    - **Why:** Many networks utilise the idea of a bottleneck, even beyond simple autoencoders (which are nothing more than a bottleneck in structure). Compressing data through a small encoding layer encourages the network to extract the most distinguishing features from the data. This is used in a number of different ways. The encoding itself can be used to represent the data in a unique and low dimensional form. Or the second half of the network can use the information from skip connections and the encoding to infer high level information about the data to solve problems.
- **Recurrence**
    - **What:** A network layer which takes as input some data and a ‘state’ vector, and produces a new state vector alongside its other output.
    - **Why:** The state vector represents some understanding about the state at a given time. The idea then is for the layer to take in some data and update this understanding, so that the state is different for the next time step. Without recurrence, networks have no notion of time or way to relate the data coming in now with what came before. This is necessary for sequential tasks like video or text recognition, and not necessary for static tasks such as image recognition, hence the discrepancy.
- **Attention**
    - **What:** Weighting each element of a sequence of data, according to how important (how much attention should be paid to) it, to solve a given task. Typically, the model will be outputting another sequence, and as such each element of the output with will require a different set of attention weights. The full theory behind attention is beyond the scope of this guide. It is listed here to give a brief intuition behind transformer models, which are growing in popularity and based on the principle of attention.
    - **Why:** Attention provides a way for a network to take in sequential data all at once, learning which input elements each output should pay attention to. This offers numerous benefits over recurrence, such as the ability to be processed in parallel (recurrent networks are inherently sequential so cannot be parallelised), and removing recency bias. Recency bias being the tendency of RNNs to pay more attention to the most recent sequence elements, rather than the most relevant.

Neural Network Design and Experimentation Process

1. Establish a strong baseline. The first thing to do once your data pipeline and evaluation metrics are set up, is to try out the simplest neural network design relevant to your problem. Usually this will be an MLP. The data will determine the input and output size, so for this make a simple MLP with one hidden layer. There is no magic formula to tell you how large to make this hidden layer, the key is to

experiment. Start with (input size + output size)/2 if you are unsure. Train and evaluate. This gives you a benchmark and some idea of how complex the task will be. If the MLP performs very well, you may wish to stop there. If not, move on to step 2.

2. Design a neural network specifically for your problem. How? Google it! More specifically, scour the internet for a research paper, article or competition submission that publishes a machine learning model for a problem similar to yours. All kinds of similarity are useful, working on the same data type (eg time series, video etc), or the same problem (eg anomaly detection, object recognition), but ideally both. For this to be successful, it is likely that the authors have already done a great deal of the work for us in choosing approximately the right kind of network architecture. Take this as a starting point.
    a. The closer their problem is to yours, the less we need to experiment with other architectures
    b. For research papers in particular, models may be more complicated than necessary as authors are usually proposing a novel method. We can experiment with removing the more complicated features, if they don't affect performance.
    c. If nothing relevant arises, use the baseline model we established earlier as the starting point.
3. Iteration and experimentation. Given a baseline (either from step 1 or 2) and working data pipeline, it may be surprisingly straightforward to test different ideas, so long as enough computational resources are available. Bear your initial goals in mind, don't be afraid to stop iterating when these are met even if it may be possible to squeeze slightly more performance out with further experimentation. Very small improvements to measured performance on test data may not actually translate to a significantly better model in the real world. APIs such as TensorFlow and PyTorch make adjusting model architectures as simple as stacking preset functions on top of one and other. These all come in built with these APIs, and are listed below:
    a. Regularisers: These may improve generalisation performance without changing the overall architecture.
        i. Dropout
        ii. Batch normalisation.
    b. Dimensionality Manipulations: Scale up or down dimensionality. Often comes at a cost of information loss in favour of computational efficiency.
        i. Max Pooling
    c. **Recurrent modules: for sequential data only.**
        i. LSTM
        ii. GRU
    d. **Other**
        i. Convolutions
        ii. Skip Connections

Taking the ideas of others as inspiration, try out some of these ideas and observe their effects, both individually and in combination. See the architectural principles section for a breakdown of what each of these do. Remember, simplicity is key. A useful method to avoid wasted time is to train and test the model after a given small change, one change at a time. If performance doesn't improve, don't waste any more time on changes of that kind. Whereas if you make many changes and then evaluate, you can't be sure which changes are having a positive effect.

Worked Examples

Pitfalls and Common Problems

Resources

Appendix