Lecture 1: Introduction

Roger Grosse

This series of readings forms the lecture notes for the course CSC421, “Neural Networks and Deep Learning,” for undergraduates at the University of Toronto. I’m aiming for it also to function as a stand-alone mini-textbook for self-directed learners and for students at other universities. These notes are aimed at students who have some background in basic calculus, probability theory, and linear algebra, but possibly no prior background in machine learning.

1 Motivation

1.1 Why machine learning?

Think about some of the things we do effortlessly on a day-to-day basis: visually recognize people, places and things, pick up objects, understand spoken language, and so on. How would you program a machine to do these things? Unfortunately, it’s hard to give a step-by-step program, since we have very little introspective awareness of the workings of our minds. How do you recognize your best friend? Exactly which facial features do you pick up on? AI researchers tried for decades to come up with computational procedures for these sorts of tasks, and it proved frustratingly difficult.

Machine learning takes a different approach: collect lots of data, and have an algorithm *automatically* figure out a good behavior from the data. If you’re trying to write a program to distinguish different categories of objects (tree, dog, etc.), you might first collect a dataset of images of each kind of object, and then use a machine learning algorithm to train a model (such as a neural network) to classify an image as one category or another. Maybe it will learn to see in a way analogous to the human visual system, or maybe it will come up with a different approach altogether. Either way, the whole process can be much easier than specifying everything by hand.

Aside from being easier, there are lots of other reasons we might want to use machine learning to solve a given problem:

- A system might need to adapt to a changing environment. For instance, spammers are constantly trying to figure out ways to trick our e-mail spam classifiers, so the classification algorithms will need to constantly adapt.

- A learning algorithm might be able to perform *better* than its human programmers. Learning algorithms have become world champions at a variety of games, from checkers to chess to Go. This would be impossible if the programs were only doing what they were explicitly told to.

- We may want an algorithm to behave autonomously for privacy or fairness reasons, such as with ranking search results or targeting ads.

Here are just a few important applications where machine learning algorithms are regularly deployed:

- Detecting credit card fraud
- Determining when to apply a C-section
- Transcribing human speech
- Recognizing faces
- Robots learning complex behaviors

1.2 How is machine learning different from statistics?

A lot of the algorithms we cover in this course originally came from statistics: linear regression, principal component analysis (PCA), maximum likelihood estimation, Bayesian parameter estimation, and Expectation-Maximization (EM). (Statisticians got there first because we had data before we had computers.) Much of machine learning, from the most basic techniques to the state-of-the-art algorithms presented at research conferences, is statistical in flavor. It's unsurprising that there should be overlap, since both fields are fundamentally concerned with the question of how to learn things from data.

What, then, is different about machine learning? Opinions will differ on this question, but if I had to offer one rule of thumb, it's this: statistics is motivated by guiding human decision making, while machine learning is motivated by autonomous agents. This means that, even when we talk about the same algorithm, practitioners in the two fields are likely to ask different questions. Statisticians might put more emphasis on being able to interpret the results of an algorithm, or being able to rigorously determine whether a certain observed pattern might have just happened by chance. Machine learning practitioners might put more emphasis on algorithms that can perform well in a variety of situations without human intervention. This overlap in techniques, coupled with the differences in motivation, creates a lot of awkwardness as practitioners in both fields will talk past each other without realizing it.

1.3 Why a course on neural networks?

Neural networks are one particular approach to machine learning, *very* loosely inspired by how the brain processes information. A neural network is composed of a large number of *units*, each of which does very simple computations, but which produce sophisticated behaviors in aggregate. There are lots of other widely used approaches to machine learning, but this class focuses on neural networks for several reasons:

- Neural nets are becoming very widely used in the software industry. They underlie systems for speech recognition, translation, ranking search results, face recognition, sentiment analysis, image search, and many other applications. It's an important tool to know.

- There are powerful software packages like Caffe, Theano, Torch, and TensorFlow, which allow us to quickly implement sophisticated learning algorithms.
- Many of the important algorithms are much simpler to explain, compared with other subfields of machine learning. This makes it possible for undergraduates to quickly get up to speed on state-of-the-art techniques in the field.

This class is very unusual among undergrad classes, in that it covers modern research techniques, i.e. algorithms introduced in the last 5 years. It's pretty amazing that with less than a page of code, we can build learning algorithms more powerful than the best ones researchers had come up with as of 5 years ago.

In fact, these software packages make neural nets *deceptively* easy. One might wonder, if you can implement a neural net in TensorFlow using a handful of lines of code, why do we need a whole class on the subject? The answer is that the algorithms generally won't work perfectly the first time. Diagnosing and fixing the problems requires careful detective work and a sophisticated understanding of what's going on beneath the hood. In this class, we'll work from the bottom up: we'll derive the algorithms mathematically, implement them from scratch, and only then look at the out-of-the-box implementations. This will help us build up the depth of understanding we need to reason about how an algorithm is behaving.

2 Types of machine learning

I said above that in machine learning, we collect lots of data, and then train a model to learn a particular behavior from it. But what kind of data do we collect? The answer will determine what sort of learning algorithm we'll apply to any given problem. Roughly speaking, there are three different types of machine learning:

- In **supervised learning**, we have examples of the desired behavior. For instance, if we're trying to train a neural net to distinguish cars and trucks, we would collect images of cars and trucks, and label each one as a car or a truck.
- In **reinforcement learning**, we don't have examples of the behavior, but we have some method of determining how good a particular behavior was — this is known as a *reward signal.* (By analogy, think of training dogs to perform tricks.) One example would be training an agent to play video games, where the reward signal is the player's score.
- In **unsupervised learning**, we have neither labels nor a reward signal. We just have a bunch of data, and want to look for patterns in the data. For instance, maybe we have lots of examples of patients with autism, and want to identify different subtypes of the condition.

This taxonomy is a vast oversimplification, but it will still help us to organize the algorithms we cover in this course. Now let's look at some examples from each category.

2.1 Supervised learning

The majority of this course will focus on supervised learning. This is the best-understood type of machine learning, because (compared with unsupervised and reinforcement learning) supervised learning problems are much easier to assign a mathematically precise formulation that matches what one is trying to achieve. In general, one defines a **task**, where the algorithm's goal is to train a **model** which takes an **input** (such as an image) and predicts a **target** (such as the object category). One collects a dataset consisting of pairs of inputs and **labels** (i.e. true values of the target). A subset of the data, called the **training set**, is used to train the model, and a separate subset, called the **test set**, is used to measure the algorithm's performance. There are a lot of highly effective and broadly applicable supervised learning algorithms, many of which will be covered in this course.

For several decades, **image classification** has been perhaps *the* prototypical application of neural networks. In the late 1980s, the US Postal Service was interested in automatically reading handwritten zip codes, so they collected 9,298 examples of handwritten digits (0-9), given as 16 × 16 images, and labeled each one; the task is to predict the digit class from the image. This dataset is now known as the USPS Dataset[1]. In the terminology of supervised learning, we say that the input is the image, and the target is the digit class. By the late 1990s, neural networks were good enough at this task that they became regularly used to sort letters.

In the 1990s, researchers collected a similar but larger handwritten digit dataset called MNIST[2]; for decades, MNIST has served as the "fruit fly" of neural network research. I.e., even though handwritten digit classification is now considered too easy a problem to be of practical interest, MNIST has been used for almost two decades to benchmark neural net learning algorithms. Amazingly, this classic dataset continues to yield algorithmic insights which generalize to challenging problems of more practical interest.

A more challenging task is to classify full-size images into object categories, a task known as **object recognition**. The ImageNet dataset[3] consists of 14 million images of nearly 22,000 distinct object categories. A (still rather large) subset of this dataset, containing 1.2 million images in 1000 object categories, is currently one of the most important benchmarks for computer vision algorithms; this task is known as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Since 2012, all of the best-performing algorithms have been neural networks. Recently, progress on the ILSVRC has been extremely rapid, with the error rate[4] dropping from 25.7% to 5.7% over the span of a few years!

All of the above examples concerned image classification, where the goal is to predict a discrete category for each image. A closely related task is **object detection**, where the task is to identify all of the objects present in their image, as well as their locations. I.e., the input is an image, and the target is a listing of object categories together with their bounding boxes.

[1] `http://statweb.stanford.edu/~tibs/ElemStatLearn/data.html`

[2] `http://yann.lecun.com/exdb/mnist/`

[3] `http://www.image-net.org/`

[4] In particular, the top-5 error rate; the algorithm predicts 5 object categories, and gets it right if any of the 5 is correct.

Other variants include **localization**, where one is given a list of object categories and has to predict their locations, and **semantic segmentation**, where one tries to label each pixel of an image as belonging to an object category. There are a huge variety of different supervised learning problems related to image understanding, depending on exactly what one is hoping to achieve. The variety of tasks can be bewildering, but fortunately we can approach most of them using very similar principles.

Neural nets have been applied in lots of areas other than vision. Another important problem domain is language. Consider, for example, the problem of **machine translation**. The task is to translate a sentence from one language (e.g. French) to another language (e.g. English). One has available a large corpus of French sentences coupled with their English translations; a good example is the proceedings of the Canadian Parliament. Observe that this task is more complex than image classification, in that the target is an entire sentence. Observe also that there generally won't be a unique best translation, so it may be preferable for the algorithm to return a *probability distribution* over possible translations, rather than a single translation. This ambiguity also makes evaluation difficult, since one needs to distinguish almost-correct translations from completely incorrect ones.

The general category of supervised learning problem where the inputs and targets are both sequences is known as **sequence-to-sequence learning**. The sequences need not be of the same type. An important example is **speech recognition**, where one is given a speech waveform and wants to produce a transcription of what was said. Neural networks led to dramatic advances in speech recognition around 2010, and form the basis of all of the modern systems. **Caption generation** is a task which combines vision and language understanding; here the task is to take an image and return a textual description of the image. The most successful approaches are based on neural nets. Caption generation is far from a solved problem, and the systems can be fun to experiment with, not least because of their entertaining errors.[5]

2.2 Reinforcement Learning

The second type of learning problem is reinforcement learning. Here, one doesn't have labels of the correct behavior, but instead has a way of quantitatively evaluating how good a behavior was; this is known as the **reward signal**. Reinforcement learning problems generally involve an **agent** situated in an **environment**. In each time step, the agent has available a set of **actions** which (either deterministically or stochastically) affect the **state** of the agent and the environment. The goal is to learn a **policy**, determining which action to perform depending on the state, in order to achieve has high a reward as possible on average.

Throughout the history of AI, a lot of progress has been driven by game playing. Over the years, AIs have come to defeat human champions in board games of increasing complexity, including backgammon, checkers, chess, and Go. In the case of Go, the success was achieved by a neural network called AlphaGo. Most of these games involve playing against an opponent,

[5] `http://deeplearning.cs.toronto.edu/i2t`

or **adversary**; this adversarial setting is beyond the scope of this class. However, single-player games can be formulated as reinforcement learning problems. For instance, we will look at the example of training an agent to play classic Atari games. The agent observes the pixels on the screen, has a set of actions corresponding to the controller buttons, and receives rewards corresponding to the score of the game. Neural net algorithms have outperformed humans on many games, in the sense of being able to achieve a high score in a short period of time.

2.3 Unsupervised Learning

The third type of machine learning, where one has neither labels of the correct behavior nor a reward signal, is known as unsupervised learning. Here, one simply has a collection of data and is interested in finding patterns in the data. We will just barely touch upon unsupervised learning in this class, because compared with supervised and reinforcement learning, the principles are less well understood, the algorithms are more mathematically involved, and one must account for a lot more domain-specific structure.

One of the most important types of unsupervised learning is **distribution modeling**, where one has an unlabeled dataset (such as a collection of images or sentences), and the goal is to learn a probability distribution which matches the dataset as closely as possible. In principle, one should be able to **generate from**, or draw samples from, the distribution, and those samples should be indistinguishable from the original data. Sometimes we care about the samples themselves, e.g. if we want to generate images of textures for graphics applications. Another important use of distribution models is to resolve ambiguities; for instance, in speech recognition, "recognize speech" may sound very similar to "wreck a nice beach," but a good distribution model ought to be able to tell us that the former is a more likely explanation than the latter.

Another important use of unsupervised learning is to recover **latent structure**, or high-level explanations that yield insight into the structure underlying the data. One important example is **clustering**, where one is interested in dividing a set of data points into **clusters**, where data points assigned to the same cluster are similar, and data points assigned to different clusters are dissimilar. Much fancier models are possible as well. For instance, a biology lab was running behavior genetics experiments on mice, and wanted to automatically analyze videos of mice to determine whether one genetic variant was more likely to engage in a particular behavior than another variant. If experts had explicitly labeled different behaviors, this would be a supervised learning problem; however, the lab avoided doing this because it would have introduced human biases into the interpretation. Instead, they ran an unsupervised learning algorithm to automatically analyze mouse videos and group them into different categories of behaviors.

3 Neural nets and the brain

The **neuron** is the basic unit of processing in the brain. It has a broad, branching tree of **dendrites**, which receive chemical signals from other neurons at junctions called **synapses**, and convert these into electrical signals.

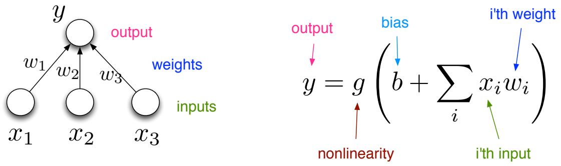

Figure 1: Simplified neuron-like processing unit.

The dendrites integrate these electrical signals in complex, nonlinear ways, and if the combined signal is strong enough, the neuron generates an **action potential**. This is an electrical signal that's propagated down the neuron's **axon**, which eventually causes the neuron to release chemical signals at its synapses with other neurons. Those neurons then integrate their incoming signals, and so on.

In machine learning, we abstract away nearly all of this complexity, and use an extremely simplified model of a neuron shown in Figure [[]]. This neuron has a set of incoming **connections** from other neurons, each with an associated strength, or **weight**. It computes a value, called the **pre-activation**, which is the sum of the incoming signals times their weights:

$$z = \sum_j w_j x_j + b.$$

The scalar value b, called a **bias**, determines the neuron's activation in the absence of inputs. The pre-activation is passed through a **nonlinearity** ϕ (also called an **activation function**) to compute the **activation** $a = \phi(z)$. Examples of nonlinearities include the **logistic sigmoid**

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

and **linear rectification**

$$\phi(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0. \end{cases}$$

In summary, the activation is computed as

$$a = \phi\left(\sum_j w_j x_j + b\right).$$

That's it. That's all that our idealized neurons do. Note that the whole idea of a continuous-valued activation is biologically unrealistic, since a real neuron's action potentials are an all-or-nothing phenomenon: either they happen or they don't, and they do not vary in strength. The continuous-valued activation is sometimes thought of as representing a "firing rate," but mostly we just ignore the whole issue and don't even think about the relationships with biology. From now on, we'll refer to these idealized neurons using the more scientifically neutral term **units**, rather than neurons.

If the relationship with biology seems strained, it gets even worse when we talk about learning, i.e. adapting the weights of the neurons. Most modern neural networks are trained using a procedure called **backpropagation**, where each neuron propagates error signals backwards through its incoming connections. Nothing analogous has been observed in actual biological neurons. There have been some creative proposals for how biological neurons might implement something like backpropagation, but for the most part we just ignore the issue of whether our neural nets are biologically realistic, and simply try to get the best performance we can out of the tools we have. (There is a separate field called theoretical neuroscience, which builds much more accurate models of neurons, towards the goal of understanding better how the brain works. This field has produced lots of interesting insights, and has achieved accurate quantitative models of some neural systems, but so far there doesn't appear to be much practical benefit to using more realistic neuronal models in machine learning systems.)

However, neural networks do share one important commonality with the brain: they consist of a very large number of computational units, each of which performs a rather simple set of operations, but which in aggregate produce very sophisticated and complex behaviors. Most of the models we'll discuss in this course are simply large collections of units, each of which computes a linear function followed by a nonlinearity.

Another analogy with the brain is worth pointing out: the brain is organized into hierarchies of processing, where different brain regions encode information at different levels of abstraction. Information processing starts at the retina of the eye, where neurons compute simple center-surround functions of their inputs. Signals are passed to the primary visual cortex, where (to vastly oversimplify things) cells detect simple image features such as edges. Information is passed through several additional "layers" of processing, each one taking place in a different brain region, until the information reaches areas of the cortex which encode things at a high level of abstraction. For instance, individual neurons in the infero-temporal cortex have been shown (again, vastly ovsersimplifying) to encode the identities of objects.

In summary, visual information is processed in a series of layers of increasing abstraction. This inspired machine learning researchers to build neural networks which are many layers deep, in hopes that they would learn analogous representations where higher layers represent increasingly abstract features. In the last 5 years or so, very deep networks have indeed been found to achieve startlingly good performance on a wide variety of problems in vision and other application areas; for this reason, the research area of neural networks is often referred to as **deep learning**. There is some circumstantial evidence that deep networks learn hierarchical representations, but this is notoriously difficult to analyze rigorously.

4 Software

There are a lot of software tools that make it easy to build powerful and sophisticated neural nets. In this course, we will use the programming language **Python**, a friendly but powerful high-level language which is widely used both in introductory programming courses and a wide variety of production systems. Because Python is an interpreted language, executing a line of Python code is very slow, perhaps hundreds of times slower than the C equivalent. Therefore, we never write algorithms directly using for-loops in Python. Instead, we **vectorize** the algorithms by expressing them in terms of operations on matrices and vectors; those operations are implemented in an efficient low-level language such as C or Fortran. This allows a large number of computational operations to be performed with minimal interpreter overhead. In this course, we will use the **NumPy** library, which provides an efficient and easy-to-use array abstraction in Python.

Ten years ago, most neural networks were implemented directly on top of a linear algebra framework like NumPy, or perhaps a lower level programming language when efficiency was especially critical. More recently, a variety of powerful neural net frameworks have been developed, including **Torch**, **Caffe**, **Theano**, **TensorFlow**, and **PyTorch**. These frameworks make it easy to quickly implement a sophisticated neural net model. Here are some of the features provided by some or all of these frameworks (we'll use TensorFlow as an example):

- **Automatic differentiation.** If one implements a neural net directly on top of NumPy, much of the implementational work involves writing procedures to compute derivatives. TensorFlow automatically constructs routines for computing derivatives which are generally at least as efficient as the ones we would have written by hand.

- **Compiling computation graphs.** If we implement a network in NumPy, a lot of time is wasted allocating and deallocating memory for matrices. TensorFlow takes a different approach: you first build a graph defining the network's computation, and TensorFlow figures out an efficient strategy for performing those computations. It handles memory efficiently and performs some other code optimizations.

- **Libraries of algorithms and network primitives.** Lots of different neural net primitives and training algorithms have been proposed in the research literature, and many of these are made available as black boxes in TensorFlow. This makes it easy to iterate with different choices of network architecture and training algorithm.

- **GPU support.** While NumPy is much faster than raw Python, it's not nearly fast enough for modern neural nets. Because neural nets consist of a large collection of simple processing units, they naturally lend themselves to parallel computation. **Graphics processing units (GPUs)** are a particular parallel architecture which has been especially powerful in training neural nets. It can be a huge pain to write GPU routines at a low level, but TensorFlow provides an easy interface so that the same code can run on either a CPU or a GPU.

For this course, we'll use two neural net frameworks. The first is **Autograd**, a lightweight automatic differentiation library. It is simple enough that you will be able to understand how it is implemented; while it is missing many of the key features of PyTorch or TensorFlow, it provides a useful mental model for reasoning about those frameworks.

For roughly the second half of the course, we will use **PyTorch**, a powerful and widely used neural net framework. It's not quite as popular as

TensorFlow, but we think it is easier to learn. But once you are done with this course, you should find it pretty easy to pick up any of the other frameworks.