VERIFIED DEEP LEARNING

AWS ALBARGHOUTHI

IN PROGRESS; DO NOT CIRCULATE

LAST UPDATED: JANUARY 22, 2020

About This Book

Why This Book

I believe that deep learning is here to stay and that we have only scratched the surface of what neural networks can actually do. The line between software 1.0 (that is, manually written code) and software 2.0 (learned neural networks) is getting fuzzier and fuzzier, and neural networks are participating in safety-critical, security-critical, and socially-critical tasks. Think, for example, healthcare, self-driving cars, malware detection, etc. But neural networks are fragile and so we need to prove that they are well-behaved when applied in critical settings.

Over the past few decades, the formal methods community has developed a plethora of techniques for automatically proving properties of programs, and, well, neural networks are programs. So there is a great opportunity to port verification ideas to the software 2.0 setting. This book offers the first introduction of foundational ideas from automated verification as applied to deep neural networks and deep learning. I hope that it will inspire verification researchers to explore correctness in deep learning and deep learning researchers to adopt verification technologies.

Who Is This Book For

Given that the book's subject matter sits at the intersection of two pretty much disparate areas of computer science, one of my main design goals is to make it as self-contained as possible. This way the book can serve as an introduction to the field for first-year graduate students even if they have not been exposed to deep learning or verification.

What Does This Book Cover

The book is divided into four parts:

Part 1 defines neural networks as data-flow graphs of operators over real-valued inputs. This formulation will serve as our basis for the rest of the book. Additionally, we will survey a number of correctness properties that are desirable of neural networks and place them in a formal framework.

Part 2 discusses *constraint-based* techniques for verification. As the name suggests, we construct a system of constraints and solve it to prove (or disprove) that a neural network satisfies some properties.

Part 3 discusses *abstraction-based* techniques for verification. Instead of executing a neural network on a single input, we can actually execute it on an *infinite* set and show that all of those inputs satisfy desirable correctness properties.

Part 4 Finally, we will discuss verification technology as applied to deep reinforcement learning tasks, where neural networks are used as controllers in a dynamical system.

Parts 2 and 3 are disjoint; the reader can go directly from Part 1 to Part 3.

Table of Contents

Part I

Neural Networks & Correctness

Chapter 1
A New Beginning

> He had become so caught up in building sentences that he had almost forgotten the barbaric days when thinking was like a splash of color landing on a page.
>
> —Edward St. Aubyn, *Mother's Milk*

1.1 It Starts With Turing

This book is about *verifying* that a *neural network* behaves according to some set of desirable properties. These fields of study, verification and neural networks, have been two distinct areas of computing research with no bridges between them, until very recently. Interestingly, however, both fields trace their genesis to a two-year period of Alan Turing's tragically short life.

In 1949, Turing wrote a little-known paper titled *Checking a Large Routine*. It was a truly forward-looking piece of work. In it, Turing asks how can we prove that the programs we write do what they are supposed to do? Then, he proceeds to provide a proof of correctness of a program implementing the factorial function. Specifically, Turing proved that his little piece of code always terminates and always produces the factorial of its input. The proof is elegant; it breaks down the program into single instructions, proves a lemma for every instruction, and finally stitches the lemmas together to prove correctness of the full program. Until this day, proofs of programs very much

Quote found in William Finnegan's *Barbarian Days*.

follow Turing's proof style from 1949. And, as we shall see in this book, proofs of neural networks will, too.

Just a year before Turing's proof of correctness of factorial, in 1948, Turing wrote a perhaps even more farsighted paper, *Intelligent Machinery*, in which he proposed *unorganized machines*. These machines, Turing argued, mimic the infant human cortex, and he showed how they can *learn* using what we now call a genetic algorithm. Unorganized machines are a very simple form of what we now know as neural networks.

1.2 The Rise of Deep Learning

The topic of training neural networks continued to be studied since Turing's 1948 paper. But it only recently exploded in popularity, thanks to a combination algorithmic developments, hardware developments, and a flood of data for training.

Modern neural networks are called *deep* neural networks, and the approach to training these neural networks is *deep learning*. Deep learning has enabled incredible improvements in complex computing tasks, most notably in computer vision and natural language processing, for example, in recognizing objects and people in an image and translating between languages. And, everyday, a growing research community is exploring ways to extend and apply deep learning to more challenging problems, from music generation to proving mathematical theorems.

The advances in deep learning have changed the way we think of what software is, what it can do, and how we build it. Modern software is increasingly becoming a menagerie of traditional, manually written code and automatically trained—sometimes constantly learning—neural networks. But deep neural networks can be fragile and produce unexpected results. As deep learning becomes used more and more in sensitive settings, like autonomous cars, it is imperative that we verify these systems and provide formal guarantees on their behavior. Luckily, we have decades of research on program verification that we can build upon, but what exactly do we verify?

1.3 What do We Expect of Neural Networks?

Remember Turing's proof of correctness of factorial? Turing was concerned that we will be programming computers to perform mathematical operations, but we could be getting them wrong. So in his proof he showed that his implementation of factorial is indeed equivalent to the mathematical definition. This notion of program correctness is known as *functional correctness*, meaning that a program is a faithful implementation of some mathematical function. Functional correctness is incredibly important in many settings—think of the disastrous effects of a buggy implementation of a cryptographic primitive.

In the land of deep learning, proving functional correctness is an unrealistic task. What does it mean to correctly recognize cats in an image or correctly translate English to Hindi? We cannot mathematically define these tasks. The whole point of using deep learning to do these tasks is because we cannot mathematically capture what exactly they entail.

So what now? Is verification out of the question for deep neural networks? No! While we cannot precisely capture what a deep neural network should do, we can often characterize some of its desirable or undesirable properties. Let us look at some examples of such properties.

Robustness

The most-studied correctness property of neural networks is *robustness*, because it is generic in nature and deep learning models are infamous for their fragility. Robustness means that small perturbations to inputs should not result in changes to the output of the neural network. For example, changing a small number of pixels in my photo should not make the network think that I am a cupboard instead of a person, or adding inaudible noise to a recording of my lecture should not make the network think it is a lecture about the Ming dynasty in the 15th century. Funny examples aside, lack of robustness can be a safety and security risk. Take, for instance, an autonomous vehicle following traffic signs using cameras. It has been shown that a light touch of vandalism to the stop sign can cause the vehicle to miss it, potentially causing an accident. Or consider the case of a neural network for detecting malware.

We do not want a minor tweak to the malware's binary to cause the detector to suddenly deem it safe to install.

Safety

Safety is a broad class of correctness properties stipulating that a program should not get to a *bad state*. The definition of *bad* depends on the task at hand. Consider a neural-network-operated robot working in a some kind of plant; we might be interested in ensuring that the robot does not exceed certain speed limits, to avoid endangering human workers, or that it does not go to a dangerous part of the plant. Another well-studied example is a neural network implementing a collision avoidance system for aircrafts. One property of interest is that if an intruding aircraft is approaching from the left, the neural network should decide to turn the aircraft right.

Consistency

Neural networks learn about our world via examples, like images. As such, they may sometimes miss basic axioms, like physical laws, and assumptions about realistic scenarios. For instance, a neural network recognizing objects in an image and their relationships might say that object A is to the on top of object B, B is on top of C, and C is on top of A. But this cannot be!

For another example, consider a neural network tracking players on the soccer field using a camera. It should not in one frame of video say that Ronaldo is on the right side of the pitch and then in the next frame say that Ronaldo is on the left side of the pitch—Ronaldo is fast, yes, but he has slowed down in the last couple of seasons.

Looking Ahead

I hope that I have convinced you of the importance of verifying properties of neural networks. In the next two chapters, we will formally define what neural networks look like (hint: they are ugly programs) and then build a language for formally specifying correctness properties of neural networks, paving the way for verification algorithms to prove these properties.

Chapter 2
Neural Networks as Graphs

There is no rigorous definition of what deep learning is and what it is not. In fact, at the time of writing this, there is a raging debate in the artificial intelligence community about a clear definition. In this chapter, we will define neural networks generically as graphs of operations over real numbers. In practice, the shape of those graphs, called the *architecture*, is not arbitrary: Researchers and practitioners carefully construct new architectures to suit various tasks. For example, neural networks for image recognition typically look different from those for natural language tasks.

First, we will informally introduce graphs and look at some popular architectures. Then, we will formally define graphs and their semantics.

2.1 **The Neural Building Blocks**

A neural network is a graph where each node performs an operation. Overall, the graph represents a function from real numbers to real numbers, that is, $\mathbb{R}^n \to \mathbb{R}^m$. Consider the following very simple graph. The red node is an

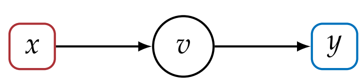


Figure 2.1 A very simple network

input node; it just passes input x, a real number, to node v. Node v performs some operation on x and spits out a value that goes to the *output* node y. For example, v might simply return $x+1$, which we will denote as the function

$f_v : \mathbb{R} \to \mathbb{R}$:

$$f_v(x) = 2x + 1$$

In our model, the output node may also perform some operation, for example,

$$f_y(x) = \max(0, x)$$

Taken together, this simple graph encodes the following function $f : \mathbb{R} \to \mathbb{R}$:

$$f(x) = f_y(f_v(x)) = \max(0, 2x + 1)$$

Transformations and Activations

The function f_v is an *affine transformation*. Simply, it multiplies inputs by constant values (in this case, $2x$) and adds constant values (1). The function f_y is an *activation* function, because it turns on or off. When its input is negative, f_y outputs 0, otherwise it outputs its input. Specifically, f_y is called a *rectified linear unit* (ReLU), and it is a very popular activation function in modern deep neural networks.

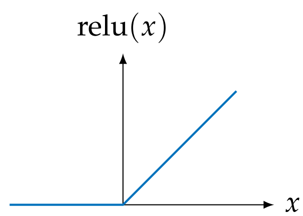


Figure 2.2 Rectified linear unit

There are other popular activation functions, for example, sigmoid,

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

whose output is bounded between 0 and 1, as shown in Figure 2.3.

Often, in the literature and practice, the affine transformation and the activation function are combined into a single operation. Our graph model of neural networks can capture that, but we usually prefer to distribute the two operations on two different nodes of the graph as it will simplify our life in later chapters when we start analyzing those graphs.

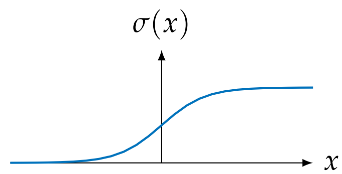


Figure 2.3 Sigmoid activation function

Universal approximation

What is so special about these activation functions? The short answer is they work in practice, in that they result in neural networks that are able to learn complex tasks. It is also very interesting to point out that you can construct a neural network comprised of ReLUs or sigmoids and affine transformations to approximate any function. This is known as the *universal approximation theorem*, and in fact the result is way more general than ReLUs and sigmoids—nearly any activation function you can think of works, as long as it is not polynomial!

2.2 Layers and Layers and Layers

In general, a neural network can be a crazy graph, with nodes and arrows pointing all over the place. In practice, networks are usually *layered*. Take the graph in Figure 2.4. Here we have 3 inputs and 3 outputs, $\mathbb{R}^3 \to \mathbb{R}^3$.

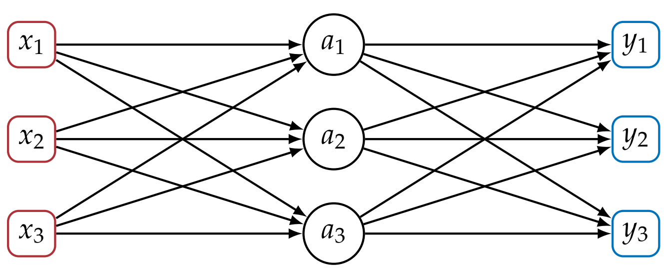


Figure 2.4 A multilayer perceptron

Notice that the nodes of the graph form layers, the input layer, the output layer, and the layer in the middle which is called the *hidden* layer. This form of graph—or architecture—has the grandiose name of *multilayer perceptron* (MLP). Usually, we have a bunch of hidden layers in an MLP, like in Figure 2.5. Layers in an MLP are called *fully connected* layers, since each node

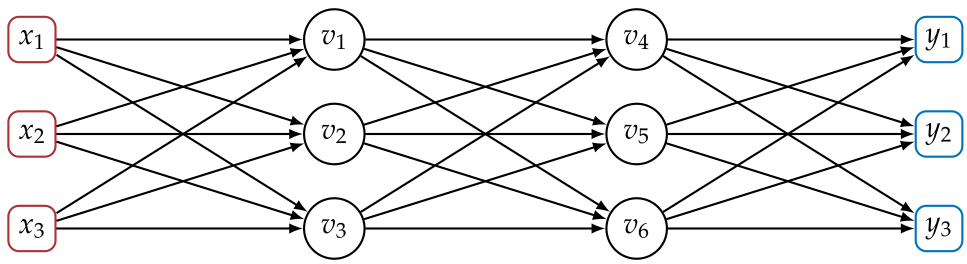


Figure 2.5 A multilayer perceptron with two hidden layers

receives all outputs from the preceding layer.

When we are doing classification, the output layer of the MLP represents the probability of each class, for example, y_1 is the probability of the input being a chair, y_2 is the probability of a TV, and y_3 of a couch. To ensure that the probabilities are normalized, that is, between 0 and 1 and sum up to 1, the final layer employs a *softmax* function. Softmax, generically, looks like this for an output node y_i, where n is the number of classes:

$$f_{y_i}(x_1, \ldots, x_n) = \frac{\exp(x_i)}{\sum_{k=1}^{n} \exp(x_k)}$$

To visualize why this actually works, please see Nielsen (2018, Chapter 3).

2.3 **Convolutional Layers**

Another kind of layer that you will find in a neural network is a *convolutional* layer. This kind of layer is widely used in computer vision tasks, but also has uses in natural language processing. The rough intuition is that if you are looking at an image, you want to scan it looking for patterns—the same thing is true of sentences in natural language. The convolutional layer gives you that: it defines an operation, a *kernel*, that is applied to every region of pixels in an image or every sequence of words in a sentence. For illustration, let us consider an input layer of size 4, perhaps each input defines a word in a 4-word sentence, as shown in Figure 2.6. Here we have a kernel, nodes v_i, that is applied to every pair of consecutive words, $(x_1, x_2), (x_2, x_3)$, and (x_3, x_4). We say that this kernel has size 2, since it takes an input in $\mathbb{R}^2$. This kernel is 1-dimensional, since its input is a vector of real numbers. In practice, we work with 2-dimensional kernels or more; for instance, to scan blocks of pixels of a gray scale image where every pixel is a real number, we can use kernels that are functions in $\mathbb{R}^{10\times 10} \to \mathbb{R}$, meaning that the kernel is applied to every 10×10 sub-image in the input.

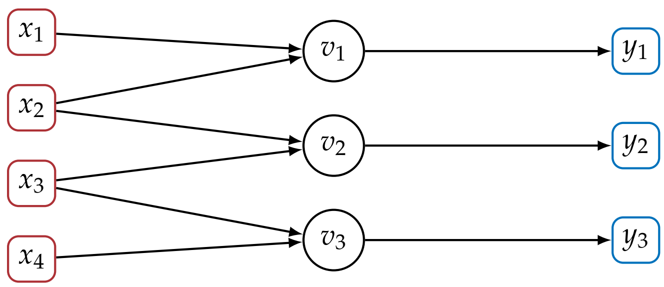


Figure 2.6 1-dimensional convolution

Typically, a *convolutional neural network* will apply a bunch of kernels to an input—and many layers of them—and aggregate (*pool*) the information from each kernel. We will formally define these operations in later chapters when we verify properties of such networks.

2.4 **Where are the Loops?**

All the neural networks we have seen so far seem to be a composition of a number mathematical functions, one after the other. So what about loops? Can we have loops in neural networks? In practice, neural network graphs are really just directed acyclic graphs (DAGs). This makes training the neural network possible using the *backpropagation* algorithm.

That said, there are classes of neural networks that appear to have loops, but they are very simple, in the sense that the number of iterations of the loop is just the size of the input. *Recurrent neural networks* (RNNs) is the canonical class of such networks, which are usually used for sequence data, like text. You will often see the graph of an RNN rendered as follows, with the self loop on node v.

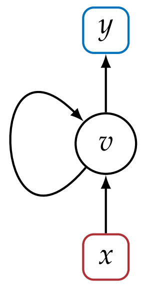


Figure 2.7 Recurrent neural network

Effectively, this graph represents an infinite family of acyclic graphs that unroll this loop a finite number of times. For example, the following is an unrolling of length 3. Notice that this is an acyclic graph that takes 3 inputs. The idea is that if you receive a sentence, say, with n words, you unroll the RNN to length n and apply it to the sentence.

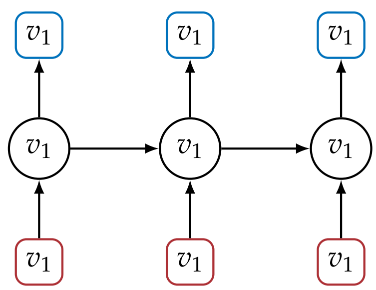


Figure 2.8 Unrolled recurrent neural network

Thinking of it through a programming lens, given an input, we can easily statically determine—i.e., without executing the network—how many loop iterations it will require. This is in contrast to, say, a program where the number of loop iterations is a complex function of its input, and therefore we do not know how many loop iterations it will take until we actually run it. With this in mind, in what follows, we will formalize neural networks as acyclic graphs.

2.5 **Structure and Semantics of Networks**

We are done with looking at pretty graphs. Let us now look at pretty symbols. We will now formally define graphs and discuss some of their properties.

Networks as DAGs

A neural network is a directed acyclic graph $G = (V, E)$, where

- V is a finite set of nodes.
- $E \subseteq V \times V$ is a set of edges.
- $V^{\mathsf{in}} \subseteq V$ is a non-empty set of input nodes.
- $V^{\mathsf{o}} \subset V$ is a non-empty set of output nodes.
- Each non-input node v is associated with a function $f_v : \mathbb{R}^n \to \mathbb{R}$, where n is the number of edges whose target is v. Notice that we assume, for simplicity but without loss of generality, that a node v only outputs a single real value. The vector of real values $\mathbb{R}^n$ that v takes as input is all the outputs of nodes v' such that $(v', v) \in E$.

To make sure that a graph G does not have any dangling nodes and that semantics are clearly defined, we will assume the following structural properties:

- All nodes are reachable, via directed edges, from some input node.
- Every node can reach an output node.
- There is fixed total ordering on edges E and another one on nodes V.

Semantics of DAGs

A network $G = (V, E)$ defines a function in $\mathbb{R}^n \to \mathbb{R}^m$ where

$$n = |V^{\mathsf{in}}| \qquad m = |V^{\mathsf{o}}|$$

That is, G maps the values of the input nodes to those of the output nodes.

Specifically, for every non-input node $v \in V$, we recursively define the value in $\mathbb{R}$ that it produces as follows. Let $(v_1, v), \ldots, (v_n, v)$ be an ordered sequence of all edges whose target is v. Then,

$$\mathsf{out}(v) = f_v(x_1, \ldots, x_n)$$

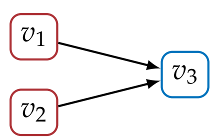
where $x_i = \mathsf{out}(v_i)$, for $i \in [1, n]$.

The base case of this definition is input nodes, since they have no edges incident on them. Suppose we are given an input $\boldsymbol{x} \in \mathbb{R}^n$, where we will use $\boldsymbol{x}$ to denote a vector and x_i to denote its ith element. Let $v_1, \ldots, v_n$ be an ordered sequence of all input nodes. Then,

$$\mathsf{out}(v_i) = x_i$$

A simple example

Let us look at an example graph G



We have $V^{\mathsf{in}} = \{v_1, v_2\}$ and $V^{\mathsf{o}} = \{v_3\}$. Now assume that

$$f_{v_3}(x_1, x_2) = x_1 + x_2$$

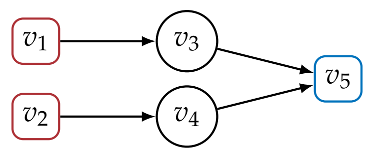
and that we are given the input vector $(11, 79)$ to the network, where node v_1 gets the value 11 and v_2 the value 79. Then, we have

$$\begin{aligned}
&\mathsf{out}(v_1) = 11 \\
&\mathsf{out}(v_2) = 79 \\
&\mathsf{out}(v_3) = \mathsf{out}(v_1) + \mathsf{out}(v_2) = 11 + 79 = 90
\end{aligned}$$

Data flow and control flow

The graphs we have defined are known in the field of program analysis as *data-flow* graphs; this is in contrast to *control-flow* graphs.[1] Control-flow graphs dictate the *order* in which operations need be performed—the flow of who has *control* of the CPU. Data-flow graphs, on the other hand, only tell us what node needs what data to perform its computation, but not how to order the computation. This is best seen through a small example.

Consider the following graph



Viewing this graph as an imperative program, one way to represent it is as follows, where $\leftarrow$ is the assignment symbol.

$$\begin{aligned}
\mathsf{out}(v_3) &\leftarrow f_{v_3}(\mathsf{out}(v_1)) \\
\mathsf{out}(v_4) &\leftarrow f_{v_4}(\mathsf{out}(v_2)) \\
\mathsf{out}(v_5) &\leftarrow f_{v_5}(\mathsf{out}(v_3), \mathsf{out}(v_4))
\end{aligned}$$

This program dictates that the value of v_3 is computed before v_4. But this need not be, as the output of one does not depend on the the other. Therefore, an equivalent implementation of the same graph can swap the first two operations:

$$\begin{aligned}
\mathsf{out}(v_4) &\leftarrow f_{v_4}(\mathsf{out}(v_2)) \\
\mathsf{out}(v_3) &\leftarrow f_{v_3}(\mathsf{out}(v_1)) \\
\mathsf{out}(v_5) &\leftarrow f_{v_5}(\mathsf{out}(v_3), \mathsf{out}(v_4))
\end{aligned}$$

Formally, we can compute the values $\mathsf{out}(\cdot)$ in any *topological* ordering of graph nodes. This ensures that all inputs of a node are computed before its own operation is performed.

[1] In deep learning frameworks like TensorFlow, they call graphs *computation graphs*.

Properties of Operations

So far, we have assumed that a node v can implement any operation f_v it wants over real numbers. In practice, to enable efficient training of neural networks, these operations need be *differentiable* or differentiable *almost everywhere*. The ReLU activation function, Figure 2.2, that we have seen earlier is differentiable almost everywhere, since at $x = 0$, there is a sharp turn in the function and the gradient is undefined.

Many of the operations we will be concerned with are *linear* or *piecewise linear*. Formally, a function $f : \mathbb{R}^n \to \mathbb{R}$ is linear if it can be defined as follows:

$$f(\boldsymbol{x}) = \sum_i c_i x_i + b$$

where $c_i, b \in \mathbb{R}$. A function is piecewise linear if it can be written in the form

$$f(\boldsymbol{x}) = \begin{cases} \sum_i c_i^1 x_i + b^1, & \boldsymbol{x} \in S_1 \\ \vdots \\ \sum_i c_i^m x_i + b^m, & \boldsymbol{x} \in S_m \end{cases}$$

where S_i are mutually disjoint subsets of $\mathbb{R}^n$ and $\cup_i S_i = \mathbb{R}^n$. ReLU, for instance, is a piecewise linear function, as it is of the form:

$$\mathrm{relu}(x) = \begin{cases} 0, & x < 0 \\ x, & x \geqslant 0 \end{cases}$$

Another important property that we will later exploit is *monotonicity*. A function $f : \mathbb{R} \to \mathbb{R}$ is monotone if for any $x \geqslant y$, we have $f(x) \geqslant f(y)$. Both activation functions we saw earlier in the chapter, ReLUs and sigmoids, are monotone. You can verify this in Figures 2.2 and 2.3: the functions never decrease with increasing values of x.

Looking Ahead

Now that we have formally defined neural networks, we are ready to pose questions about their behavior. In the next chapter, we will formally define a language for posing those questions. Then, in the chapters that follow, we will look at algorithms for answering those questions.

Chapter 3

Correctness Properties

Part II

Constraint-Based Verification

Chapter 4

Decidable Theories of First-Order Logic

Chapter 5
Encodings of Neural Networks

Chapter 6

Decision Procedures

Chapter 7

Specialized Decision Procedures

Part III

Abstraction-Based Verification

Chapter 8

Just Enough Abstract Interpretation

Chapter 9

Numerical Abstract Domains

Chapter 10

Abstract Execution of Neural Networks

Chapter 11

Abstract Deep Learning

Part IV

Verified Reinforcement Learning

Chapter 12

Neural Networks as Policies

Chapter 13

Verifying RL Policies

Chapter 14

Efficient Policy Verification

Chapter 15

Enforcing Properties in RL

Bibliography

Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2018. URL http://neuralnetworksanddeeplearning.com/.