1.3 Literature Survey

The topic of DNN testing is a fast-growing subject with new research being added every month, and as such the scope of my research has expanded a lot since the last report. This literature survey is organized as such. Macro-level concepts are introduced first, giving a brief overview of core concepts used for the Design. As we move down, more micro-level concepts are discussed. Finally, papers which explore interesting concepts but happen to be orthogonal to our work are discussed.

It is only necessary to read through the subheadings 1.3.1 -1.3.9 to gain enough context to understand the work in the methodology segment later.

1.3.1 Deep Neural Networks

DNNs are software structures consisting of layers, each layer containing of multiple neurons. Each neuron applies an activation function to the sum of its inputs (if the neurons are not on the first layer, the inputs are the output of neurons in the previous layer), and the output is passed to the neuron on the next layer. Each connection between each neuron has an associated weight and bias. The main activation functions are sigmoid, hyperbolic tangent and ReLU [12].

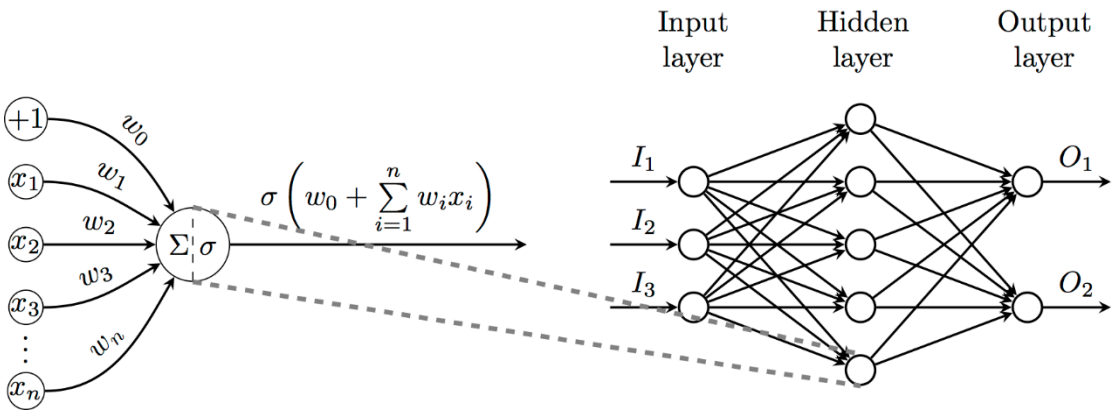

Figure 1: A simple DNN, with more detail on the neuron

The network is tested against the training dataset and evaluated using a loss function. To train the network, gradient descent using backpropagation is used [13] to minimize the loss function.

1.3.2 Adversarial Attack

Adverserial attack is a popular research topic for DNNs. The technique takes advantage of the fact that DNNs allow for easy gradient calcuulation, thereby making it easy to explore the input space for misclassified images [14] [15] [16]. Some versions may begin with a valid input, where others may begin from scratch. These attacks are difficult to defend against because they make use of inherent properties of DNNs, namely the existent of gradients.

Adverserial attack is shown to work even when the resultant adverserial image is printed, and scanned with a phone [17]. This is shown to work even when the DNN is made to not show certainty/masked (thus removing direct access gradient ascent), because adversarial inputs obtained for different models that perform a similar function also produce erroneous behavior in the masked DNN [18].

Two proposed defences are defensive distillation [19] – outputing multiple classes instead of a single output, something that doesn't quite work for autonomous steering – and adverserial training – using the generated adverserial data to train the model – an idea that is used in many other papers [10] [20].

This idea of using adverserial attack to produce "difference-inducing inputs" is quite common throughout the research in the field, and forms the core structure of many of

the papers discussed.

The core flow of this is we have a heruristic, which we use to guide a generation method, which can be additive functions inputs [21], or single pixels changes [16]. We will be discussing other generation methods later on.

1.3.3 Decision Boundaries

For neural networks that output a single output, decision boundaries between two outputs often have rougher gradients, meaning erroneous behaviors are more common. Also, models of similar function will have similar (though not identical) decision boundaries. Erroneous behaviors are thought to lie around decision boundaries. However, my research has not explored the nature of decision boundaries are like for models that have continuous outputs (like autonomous vehicle steering). It will be worth doing further research into the topic.

1.3.4 Metamorphic Relations

There is this idea that certain transforms (like image rotation, mirroring, color inversion), while they can significantly alter the image (increase the distance between the images), they do not change the structure of the data. It follows that this changed data can be used to train a new model, and this new model will be able to successfully categorize the transformed inputs. Mislabeled or dummy training data, however will not be correctly categorized when transformed, as their structure is not consistent with their labeling. Thus, we can use these to clean our datasets of dirty data. These ideas are presented in the paper by Dwarakanath et al. [22], and concepts from metamorphic testing – using the idea that certain transforms to the input should alter the output of a system – are referenced in DeepTest [20], DeepRoad [23], and others [11] [16] [24].

1.3.5 Main Behaviors Vs. Corner Case Behaviors

One dichotomy that is presented in DeepGauge [11] is main behaviors – the input space that is covered by the training data – and corner-case behavior – input space that is not covered by training data. While adversarial inputs can lie in both the spaces, different testing criterion prioritize different spaces, for example neuron coverage favors main behaviors, whereas criterion like K-multisection Neuron Coverage [11] favor corner-case behaviors. DeepHunter [25] suggests that more adversarial inputs can be found by exploring corner-case behaviors, which is something that follows logically. If a certain input/behavior set is not trained for, it seems natural that the DNN would be more likely to make a mistake.

1.3.6 Traditional Software Engineering and DNN Software Engineering

Many of the papers we will be discussing use traditional software testing criterions and techniques, then attempt to adapt the essence of these methods to DNNs. Forms of code coverage, taken from traditional programming are adapted to DNNs. Test case generation techniques, like concolic testing and fuzzing have also been adapted.

1.3.7 Neuron Coverage

The team behind DeepXplore [10] proposed a testing criterion. Adapted from the concept of "code coverage" in traditional programming, they introduced "neuron coverage" a measure of how much of the model's neurons had been used. Neuron coverage is calculated using the following method: for a certain input, if some neurons exceeds a certain activation threshold, we count them as being "activated". Over the entire test set, we see how many neurons in the DNN have been activated at least once. They show that existing methods of adversarial attack generation do not produce high neuron coverage. In using this criterion in conjunction with adversarial attacks' difference

inducing gradient ascent, we produce adversarial inputs that also maximize neuron coverage. This concept of neuron coverage is used in many proceeding papers, however, there are some key critiques. Some papers found that neuron coverage was easily satisfied with a few images [24] [25]. DeepGauge [11] posits that the neuron coverage is too abstracted and does not take into account the statistical properties of the activation function, i.e. some neurons are almost always highly activated, some are difficult to activate, and this is not reflected in the coverage.

1.3.8 Further Coverage Criterion

Many different papers have continued to propose newer criterions. DeepGauge [11] considered many multi-granular coverage criterions that were more difficult to satisfy. Other papers have also shown other coverage criterion, many inspired by existing software testing methods, like MC/DC [26], Lipschitz continuity [27] and combinatorial testing [28]. The paper DeepHunter [25] has implemented Neuron Coverage from DeepXplore [10]; as well as K-multisection Neuron Coverage, Neuron Bound. Coverage, Strong Neuron Act. Coverage, Top-k Neuron Coverage from DeepGauge [11]; and gives their assessment of these coverage criterion, namely that neuron coverage is easily satisfied, and that the other 4 from DeepGauge skew towards exploring corner-case behavior, rather than main behaviors. They proposed K-multisection Neuron Coverage as a criterion to quantify exploring main behaviors.

1.3.9 DeepGauge's Multi-Granular Coverage

The work by DeepGauge introduced the innovation of using training data to find a single neuron's maximum and minimum output, using this as a rough boundary for main function behavior and corner-case behavior. They also introduced considering the permutations of activations of neurons to give a more difficult to satisfy coverage

criterion.

1.3.10 Modified Condition/Decision Coverage Based Coverage Criterion

MC/DC is a coverage criterion that requires the following [29]:

1. Each entry and exit point is invoked
2. Each decision takes every possible outcome
3. Each condition in a decision takes every possible outcome
4. Each condition in a decision is shown to independently affect the outcome of the decision.

DeepCover [26] explored criterions based on MC/DC concepts. Sign-sign neuron coverage – looking at permutations of the signs (negative and positive) of pairs of neurons – is presented in the paper by DeepCover and is reused in TensorFuzz [24]. It is similar to the next coverage metric, combinatorial coverage.

1.3.11 Combinatorial Testing Coverage

The papers by DeepCover [26] have mentioned the use of concepts from combinatorial testing, namely in sign-sign neuron coverage, and these ideas are fully explored in the work on DeepCT by Ma et al. [28]. They reduce each neuron to being activated/de-activated. Then, they can consider the permutations of activating/de-activating groups of neurons of size k. By considering how extensively these permutations are explored, they evaluate the coverage.

The main benefit they claim is that erroneous behaviors occur when pairs of factors interact. Thus, by considering permutations of pairs of neurons, we obtain a more comprehensive look logic coverage. While their coverage criteria are harder to fulfill than neuron coverage, they do not consider the statistical properties of the neurons.

1.3.12 Lipschitz Coverage

In the DeepConcolic [27] they use Lipschitz Continuity: a form of continuity that states that the maximum slope of a continuous function is limited by a real number c. In application they use this concept to define Lipschitz Coverage: an input is Lipschitz Covered iff in an open ball of size b (a parameter), there exists no inputs which cause outputs that differ by more than c (another parameter). i.e. if the open ball centered around an input fluctuates in output greatly, then it is not Lipschitz Covered.

1.3.13 Fuzzing

Fuzzing is a technique from traditional software testing [30], which uses seeds to produce random inputs which can be used to expose erroneous behavior. These seeds can be saved so the error inducing input may be reproduced and examined. There are a few differentiators. The first is when the Fuzzer mutates existing inputs or creates inputs from scratch; second is whether they are aware of input structure or not; third is they can be white, grey, or black box [30]. Papers like TensorFuzz [24], DeepHunter [25], and DLFuzz [31] have reapplied these methods with DNNs, and incorporated the coverage criterions discussed in DeepXplore [10] and DeepGauge [11]. By mutating inputs and using coverage criterion as a heuristic, we can obtain higher coverage. Due to the small size of the perturbations, we use metamorphic relations to assume that the output should not change significantly. Thus, we can obtain adversarial inputs. One novel benefit of this method is due to its random nature, there is no inherent order in which we must process the data. This means that using a probabilistically weighted batch implementation, it is possible to parallelize the testing and increase throughput, which may be useful in operation [25].

1.3.14 Concolic Testing [27]

Concolic testing is a software testing technique that uses concrete testing (running with a concrete input) and symbolic testing (analyzing the program to find what inputs are needed to get a code block to execute).

The paper "Concolic Testing for Deep Neural Networks" discusses implementing concolic testing techniques for DNN. However, while sounding quite novel, in practice symbolic analysis is basically the gradient ascent and criterion-based heuristics used by other papers. They even reuse sign-sign coverage from MC/DC [26] and Neuron coverage from DeepXplore [10]. Their most novel contribution is a new criterion called "Lipschitz Coverage".

1.3.15 DeepCruiser or Markov Decision Process Testing [32]

An adaption of some of the ideas involving input space coverage proposed by DeepXplore [10] and DeepGauge [11] to RNNs (recurrent neural networks). Some of the main difficulties of adapting neuron coverage and similar coverage criterions is they do not cover the attention-vector used by RNNs. This paper uses the paths of the attention vectors for training data (called "traces" in the paper), and by collectively analyzing the traces, we obtain an MDP. Using this, we can test how extensively our test data explores our abstracted MDP.

While their techniques are not relevant to our work, we would like to highlight that they use analysis of how the model handles training data to quantify whether test data is within the main function body or is outside standard functionality. They also use metamorphic relations as a basis for assuming the outputs of the generated adversarial data.

1.3.16 Training Set Reduction [33]

By reducing the size of the dataset, we may train models more quickly. Considers the optimal method to cut the original dataset, including random selection, distance based selection, and loss-function based selection. It concludes that loss-function based selection is most optimal. The paper is quite orthogonal to our area of research.

1.3.17 Data Cleaning

One widely cited use of these methods is their ability to clean training data sets of mislabeled/malicious data [22]. This is a worthy application that we may be able to explore further down the line, if time allows.

1.3.18 Feature Based Adversarial Testing [34]

Using the feature extraction tool SIFT, they extract the key features of an image. They find that when single pixel perturbations are applied near the features, erroneous output is more common. This is another alternative heuristic to gradient ascent that can be to guide adversarial input finding.

1.3.19 DeepMutation or Mutation Testing [35]

An adaption of the technique of mutation testing. The program to be tested is mutated in several ways, and faults are introduced to the program. Then, the mutants and original program are subjected to the test suite, and failures to distinguish between mutant and original are noted.