Verification of Neural Networks for the masses: are our programming languages ready?

Katya Komendantskaya,
Lab for AI and Verification, HWU
www.LAIV.uk

28 May 2019

Lab for AI and Verification

- Launched in March 2019
- A “grass-route” Lab: initially launched to support MSc and PhD research in verification of AI, for AI and Robotics students
- Rapidly grew to 12 people: 5 academic staff, 4 PhD and 3 MSc students not counting a number of collaborators.

LAIV members:





Pervasive AI...

Pervasive AI...

Autonomous cars

Pervasive AI...

Autonomous cars

Smart Homes

Pervasive AI...

Autonomous cars

Smart Homes

Robotics

Pervasive AI...

Autonomous cars

Smart Homes

Robotics

Chat Bots

Pervasive AI...

Autonomous cars

Smart Homes

Robotics

Chat Bots

...and many more ...

AI is in urgent need of verification: safety, security, robustness to changing conditions and adversarial attacks, ...

LAIV Research Topics

- Verification of AI Planning languages
- Verification of Neural Networks
- Machine Learning for Verification
- ... see www.laiv.uk for more

Outline

Verification of AI and LAIV.uk

Challenges in Verification of Neural Networks

Demo: Verify Perceptron

- ...with Python and Z3
- ...with Coq
- ... with F*

Outline

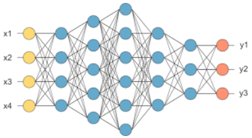
Verification of AI and LAIV.uk

Challenges in Verification of Neural Networks

Demo: Verify Perceptron

- ...with Python and Z3
- ...with Coq
- ... with F*

Neural Nets in Massive use



Used for:

- computer vision
- speech recognition
- (big) data processing
- ...

In:

- autonomous cars
- robots
- airport security
- financial applications
- . . .
- Alexa
- Google bot on mobile phones
- image recognising apps

BIG PROOF aspects?

Neural net verification is a "Big Proof" business:

- Objects are big: thousands of nodes and millions parameters to train in modern neural nets
- Automated verification can take days

BIG PROOF aspects?

Neural net verification is a “Big Proof” business:

- Objects are big: thousands of nodes and millions parameters to train in modern neural nets
- Automated verification can take days
- There is a lot of maths: linear algebra, probabilities, statistics behind machine learning
- Interactive proofs rely on mature proof infrastructure

Neural nets and verification

Should Neural net verification be different from any other kind of verification?

- In principle, – No. But there are a few specific features:
  - the object we verify (neural net) is not programmed but obtained via learning from data.
    We may be interested in either the learning algorithms or the properties of the resulting net.

Neural nets and verification

Should Neural net verification be different from any other kind of verification?

- In principle, – No. But there are a few specific features:
  - the object we verify (neural net) is not programmed but obtained via learning from data.
    We may be interested in either the learning algorithms or the properties of the resulting net.
  - its outputs have statistical/probabilistic nature
    We may be interested in verifying probabilistic properties

Neural nets and verification

Should Neural net verification be different from any other kind of verification?

- In principle, – No. But there are a few specific features:
  - the object we verify (neural net) is not programmed but obtained via learning from data.
    We may be interested in either the learning algorithms or the properties of the resulting net.
  - its outputs have statistical/probabilistic nature
    We may be interested in verifying probabilistic properties
  - we have to be able to work with real numbers, not integers or bits
    Real numbers are a challenge for provers, especially based on constructive logic

The literature splits

There are two groups of properties we may want to verify:

- General (concerning properties of learning algorithms): e.g. how well does the learning algorithm perform? do trained neural networks generalise well?
  - A. Bagnall and G. Stewart. Certifying the True Error: Machine Learning in Coq with Verified Generalisation Guarantees. AAAI 2019.
  - A. Bahrami, E. de Maria and A.Felty. Modelling and Verifying Dynamic Properties of Biological Neural Networks in Coq. 2018.

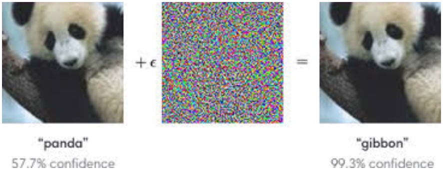
The literature splits

There are two groups of properties we may want to verify:

- General (concerning properties of learning algorithms): e.g. how well does the learning algorithm perform? do trained neural networks generalise well?
  - A. Bagnall and G. Stewart. Certifying the True Error: Machine Learning in Coq with Verified Generalisation Guarantees. AAAI 2019.
  - A. Bahrami, E. de Maria and A.Felty. Modelling and Verifying Dynamic Properties of Biological Neural Networks in Coq. 2018.
- Specific to applications (concerning neural network deployment): given this trained neural network, is it robust to adversarial attacks?
  - X. Huang and M. Kwiatkowska and S. Wang and M. Wu. Safety Verification of Deep Neural Networks. CAV (1) 2017: 3-29
  - G. Singh, T. Gehr, M. Puschel, M. T. Vechev: An abstract domain for certifying neural networks. PACMPL 3(POPL): 41:1-41:30 (2019)

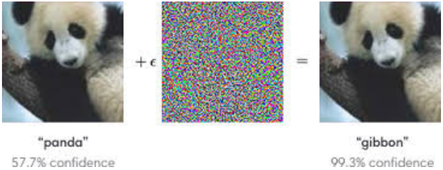
The language gap

Adversarial attacks/defences



The language gap

Adversarial attacks/defences

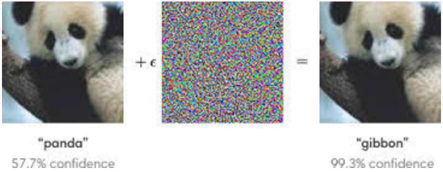


Languages: Python and SMT Solvers (Z3). Pros:

- use Python's rich infrastructure for machine learning
- automation

The language gap

Adversarial attacks/defences



Languages: Python and SMT Solvers (Z3). Pros:

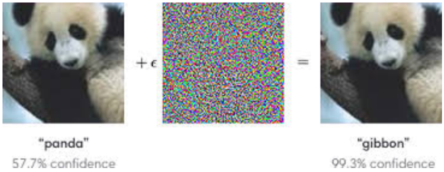
- use Python's rich infrastructure for machine learning
- automation

Proofs of general properties of neural nets

- Generalisation bounds
- Properties of network architectures
- Equality of networks
- ...

The language gap

Adversarial attacks/defences



Languages: Python and SMT Solvers (Z3). **Pros:**

- use Python's rich infrastructure for machine learning
- automation

Proofs of general properties of neural nets

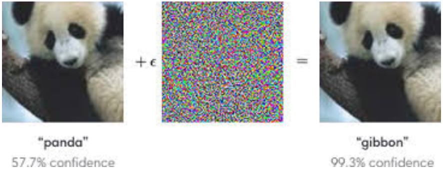
- Generalisation bounds
- Properties of network architectures
- Equality of networks
- ...

Languages: Higher-order interactive provers (e.g. Coq). **Pros:**

- Generality
- Rich proof infrastructure

The language gap

Adversarial attacks/defences



Languages: Python and SMT Solvers. **Cons:**

- proving general properties is infeasible: weak link between Z3 and Python's objects;
- fragile types

Proofs of general properties of neural nets

- Generalisation bounds
- Properties of network architectures
- Equality of networks
- ...

Languages: Higher-order interactive provers (e.g. Coq). **Cons:**

- computing with defined objects is awkward;
- less automation

...A further note on infrastructure

- Although dozens of research papers on neural net verification exist,
- There is no developed infastructure for neural net verification in either camp

...A further note on infrastructure

- Although dozens of research papers on neural net verification exist,
- There is no developed infastructure for neural net verification in either camp

Existing gap in undergraduate and postgraduate education

- demand in verification coming from students and industries
- ... cannot be currently met in mass education (no mature and easy-to-use tools/languages to use)
- *doing it on level of individual MSc projects with top students still bears challenges*

Demo

Challenges of neural network verification:

- by simple example
- as a good student may find them
- with emphasis on state-of-the-art in programming language infrastructure

... paying attention to "hammer determines the nails" effect

Outline

Verification of AI and LAIV.uk

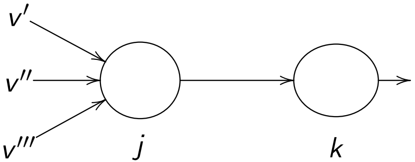
Challenges in Verification of Neural Networks

Demo: Verify Perceptron

- ...with Python and Z3
- ...with Coq
- ... with F*

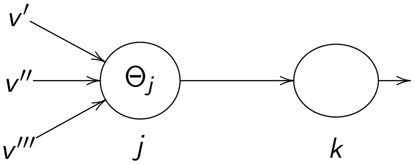
Artificial Neurons

Neuron's potential: $p_k(t) = \sum_{j=1}^{n_k} w_{kj}(t) v_j(t) - \Theta_k$
Neuron's value: $v_k(t) = \psi(p_k(t))$



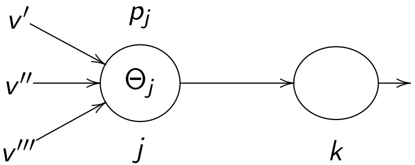
Artificial Neurons

Neuron's potential: $p_k(t) = \sum_{j=1}^{n_k} w_{kj}(t)v_j(t) - \Theta_k$
Neuron's value: $v_k(t) = \psi(p_k(t))$



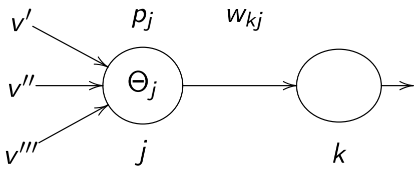
Artificial Neurons

Neuron's potential: $p_k(t) = \sum_{j=1}^{n_k} w_{kj}(t)v_j(t) - \Theta_k$
Neuron's value: $v_k(t) = \psi(p_k(t))$



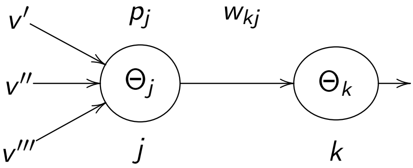
Artificial Neurons

Neuron's potential: $p_k(t) = \sum_{j=1}^{n_k} w_{kj}(t) v_j(t) - \Theta_k$
Neuron's value: $v_k(t) = \psi(p_k(t))$



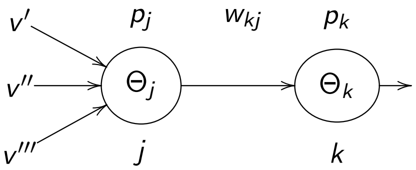
Artificial Neurons

Neuron's potential: $p_k(t) = \sum_{j=1}^{n_k} w_{kj}(t)v_j(t) - \Theta_k$
Neuron's value: $v_k(t) = \psi(p_k(t))$



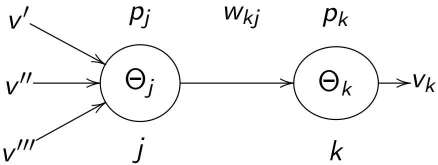
Artificial Neurons

Neuron's potential: $p_k(t) = \sum_{j=1}^{n_k} w_{kj}(t) v_j(t) - \Theta_k$
Neuron's value: $v_k(t) = \psi(p_k(t))$



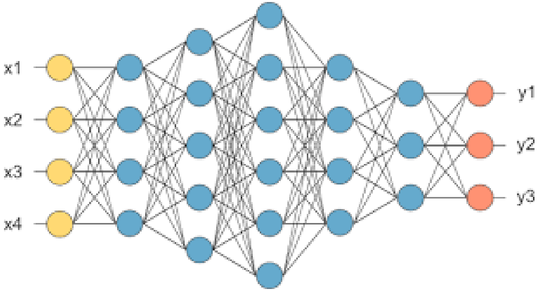
Artificial Neurons

Neuron's potential: $p_k(t) = \sum_{j=1}^{n_k} w_{kj}(t)v_j(t) - \Theta_k$
Neuron's value: $v_k(t) = \psi(p_k(t))$



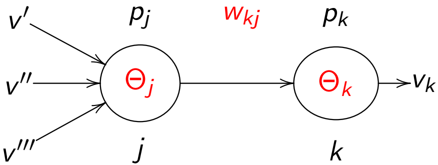
Neural Network is...

a directed graph where each node and edge has the above parameters...

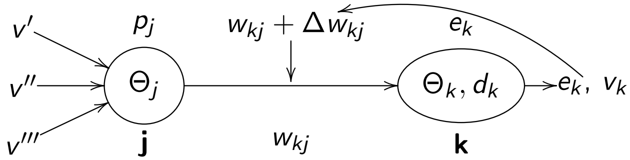


We train Neural nets by:

- adjusting the weights;
- adjusting the thresholds.

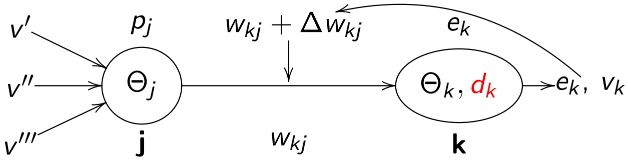


Error-Correction (Supervised) Learning



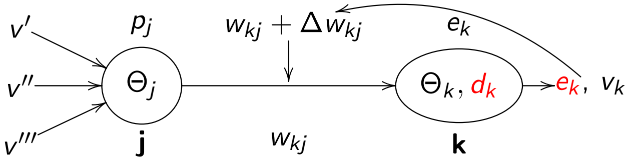
Error-Correction (Supervised) Learning

We provide a **desired response** d_k;



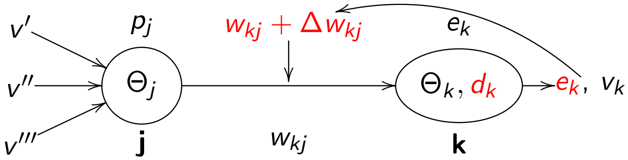
Error-Correction (Supervised) Learning

We provide a **desired response** d_k;
Error-signal: e.g. absolute error $e_k(t) = d_k(t) - v_k(t)$;



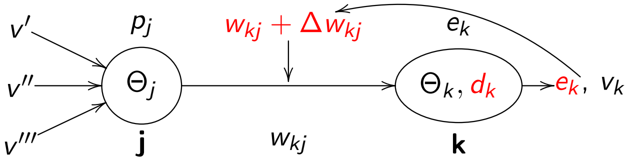
Error-Correction (Supervised) Learning

We provide a **desired response** d_k;
Error-signal: e.g. absolute error $e_k(t) = d_k(t) - v_k(t)$;
Error-correction learning rule: $\Delta w_{kj}(t) = \eta e_k(t) v_j(t)$.



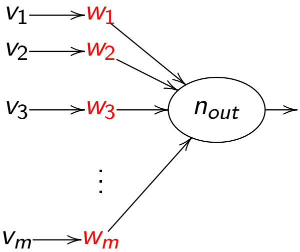
Error-Correction (Supervised) Learning

We provide a **desired response** d_k;
Error-signal: e.g. absolute error $e_k(t) = d_k(t) - v_k(t)$;
Error-correction learning rule: $\Delta w_{kj}(t) = \eta e_k(t) v_j(t)$.



This mode of learning is called *gradient descent*.

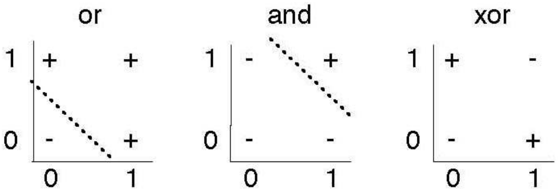
Perceptron



This network simulates the linear function:
$f(v_1, v_2, \ldots, v_m) = \psi(\theta + v_1 w_1 + v_2 w_2 + \ldots + v_m w_m)$, where ψ is whatever activation function the neuron n_{out} has.

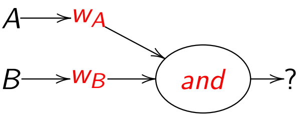
Historical uses of Neural nets: Perceptron

Neural nets doing logic [McCulloch and Pitts, 1943]:



A	B	A and B	A or B	A xor B
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

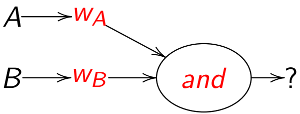
Perceptron for and



Input features and target features:

A	B	A and B
true	true	true
true	false	false
false	true	false
false	false	false

Perceptron for and

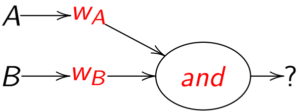


Input features and target features:

A	B	A and B
true	true	true
true	false	false
false	true	false
false	false	false

Now train the network: will it be able to learn the correct (linear) function $\theta + w_A \times A + w_B \times B$ to simulate and?

Perceptron for and



Input features and target features:

A	B	A and B
true	true	true
true	false	false
false	true	false
false	false	false

Now train the network: will it be able to learn the correct (linear) function $\theta + w_A \times A + w_B \times B$ to simulate and?
e.g. $-0{,}9 + 0{,}5 \times A + 0{,}5 \times B$

Outline

Verification of AI and LAIV.uk

Challenges in Verification of Neural Networks

Demo: Verify Perceptron

- ...with Python and Z3
- ...with Coq
- ... with F*

Python

- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.
- Used for scripting
- Most popular language in machine learning
- A huge infrastructure of machine learning libraries,
- including e.g. TensorFlow (by Google) and PyTorch (by Facebook)

Demo

Robustness Verification scenario

- Implement my Perceptron in Python
- Prove it is robust for class 1:
  - Define its robustness region: e.g. when input array contains real values in the region $\epsilon = [0{,}5; 1{,}5]$
  - define a step function ("the ladder") to generate a finite number of reals in this region (or Z3 will not terminate)
  - Prove the ladder is "covering" (using pen and paper)
  - Take the set of input matrices generated by Z3, run them through the Perceptron
  - No mis-classification? – I have proven my network robust for output 1, region ϵ and the ladder.

Robustness Verification scenario

- Implement my Perceptron in Python
- Prove it is robust for class 1:
  - Define its robustness region: e.g. when input array contains real values in the region $\epsilon = [0{,}5; 1{,}5]$
  - define a step function ("the ladder") to generate a finite number of reals in this region (or Z3 will not terminate)
  - Prove the ladder is "covering" (using pen and paper)
  - Take the set of input matrices generated by Z3, run them through the Perceptron
  - No mis-classification? – I have proven my network robust for output 1, region ϵ and the ladder.

Problems

- No direct access to Perceptron implementation from Z3
- Fragility of type conversion between Python and Z3
- Either "Testing flavour" or manual proofs are needed

Outline

Verification of AI and LAIV.uk

Challenges in Verification of Neural Networks

Demo: Verify Perceptron

- ...with Python and Z3
- ...with Coq
- ... with F*

Coq

- Needs no introduction in this audience
- Functional, dependently-typed language
- Interactive theorem prover
- Mature library infrastructure (thanks to some very famous people in this room!)

Coq verification scenario

- No black-box tricks: you define all objects exactly

```
Record Neuron := MakeNeuron {
  Output: list nat;
  Weights: list Q;
  Psi: Q;
  Theta: Q;
  Current: Q;
  Output_Bin: Bin_List Output;
  LeakRange: Qle_bool 0 Psi = true /\ Qle_bool Psi 1 = true;
  PosTheta: Qlt_bool 0 Theta = true;
  WRange: WeightInRange Weights = true
}.
```

- You prove their properties (generality limited only by imagination)

```
Lemma NextOutput_Bin_List: forall (N: Neuron) (Inputs: list nat),
Bin_List (Output N) -> Bin_List (NextOutput N Inputs::Output N).
```

Coq verification scenario

- We can even define the Perceptron computed by my Python code (with rationals not reals:)

```
Lemma Perceptron : Neuron.
Proof.
  apply (MakeNeuron [0%nat] [2#10; 2#10] 1 (2#10) 0); simpl;
      auto.
Qed.
```

Cf: my Python computation was:

```
Weights after training:  [-0.18375655  0.19388244  0.19471828]
```

Coq verification scenario

- But try computing or evaluating:

```
Definition Pp : nat :=
  NextOutput Perceptron [1%nat; 1%nat].

Compute (Pp).
```

- You get:

```
 = if
    match
      match
        match
          (let (Qnum, _) :=
             let (Output, Weights, Psi, Theta, _, _, _, _, _) :=
               Perceptron in
             Theta in
           Qnum)
        with
        | 0%Z => 0%Z
        | Z.pos x =>
            Z.pos
              ((fix Ffix (x0 x1 : positive) {struct x0} :
                   positive :=
                   match x0 with
                   | (x2~1)%positive =>
                       (fix Ffix0 (x3 x4 : positive) {struct x3} :
```

Coq verification scenario

- same problem arises when proving properties of individual networks...

```
Lemma robust: forall x y : nat, x = 1%nat -> y = 1%nat -> (NextOutput
    Perceptron [x ; y]) = 1%nat.
```

- Possibly reflecting to Booleans may help in some cases (SSR)?

Coq verification scenario

- same problem arises when proving properties of individual networks...

```
Lemma robust: forall x y : nat, x = 1%nat -> y = 1%nat -> (NextOutput
    Perceptron [x ; y]) = 1%nat.
```

- Possibly reflecting to Booleans may help in some cases (SSR)?
- If you are a machine-learning student/professional, you will ofcourse miss Python libraries for processing your big data sets, using different learning algorithms, ...
- you will miss real numbers that were hassle-free in Python

Outline

Verification of AI and LAIV.uk

Challenges in Verification of Neural Networks

Demo: Verify Perceptron

- ...with Python and Z3
- ...with Coq
- ... with F*

F*

- a general-purpose functional programming language with effects
- aimed at program verification
- puts together the automation of an SMT-backed deductive verification tool
- with the expressive power of a proof assistant based on dependent types.
- After verification, F* programs can be extracted to efficient OCaml, $F\#$, C, WASM, or ASM code.

F* to the rescue?

- You can get your reals back!

```
noeq type neuron =
| MakeNeuron :
 output: list nat
 -> weights: list real
 -> psi: real
 -> theta: real
 -> current: real
 -> output_Bin: bin_list output
 -> leakRange: (0.0R <=. psi) /\ (psi <=. of_int 1)
 -> posTheta: 0.0R <=. theta
 -> wRange: weightinrange weights
-> neuron
```

F* to the rescue?

- You can get your connection to SMT solver back!

```
val perceptron :
  neuron
 let perceptron = MakeNeuron [0] [0.194R ; 0.195R] 1.0R 0.184R
     0.0R


 let add_id_l = assert (forall m n. ( (m >=. 0.5R) /\ (n >=. 0.5R
     )) ==> (nextoutput perceptron [m ; n]) == 1)
```

F* to the rescue?

- You can get your connection to SMT solver back!

```
val perceptron :
  neuron
 let perceptron = MakeNeuron [0] [0.194R ; 0.195R] 1.0R 0.184R
     0.0R


 let add_id_l = assert (forall m n. ( (m >=. 0.5R) /\ (n >=. 0.5R
     )) ==> (nextoutput perceptron [m ; n]) == 1)
```

- The above goes via Z3 check, and is more general than we had via Python/Z3 ladder generation,
- moreover, not opaque for the Perceptron definition in F*!
- General properties are still proven smoothly

F* to the rescue?

- But Computing is still hard!
- Mixing SMT solver output of type `Unit`, and other properties (`Prop`) is hard.
- So, this lemma does not yield an automated proof via Z3:

```
val pp2: #a: Type -> Lemma ( (nextoutput perceptron [1.0R ;
    1.0R]) == 1)
let pp2 = ()

(Error) Expected expression of type "Prims.Lemma unit
(nextoutput perceptron [1.0R; 1.0R] == 1) []"; got
    expression "()" of type "unit"
```

- Still some way to go to become a widely used neural net verification tool...

Conclusions

- Demand for neural net verification is growing
- The general methodology/ tools are there (like bits in a puzzle)
- but they are not really gathered into a language ready to become an industrial or educational tool
- A language like F* is a good idea, though better integration is needed:
  - with mainstream machine-learning languages like Python
  - better interoperability between dependent types and SMT solvers (if at all possible?)

Conclusions

- Demand for neural net verification is growing
- The general methodology/ tools are there (like bits in a puzzle)
- but they are not really gathered into a language ready to become an industrial or educational tool
- A language like F* is a good idea, though better integration is needed:
    - with mainstream machine-learning languages like Python
    - better interoperability between dependent types and SMT solvers (if at all possible?)

LAIV is looking for solutions...

Thanks for your attention!