Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees

Alexander Bagnall and **Gordon Stewart**
Ohio University, Athens, OH 45701
{ab667712, gstewart}@ohio.edu

Abstract

We present MLCert, a novel system for doing practical mechanized proof of the generalization of learning procedures, bounding expected error in terms of training or test error. MLCert is mechanized in that we prove generalization bounds inside the theorem prover Coq; thus the bounds are machine checked by Coq's proof checker. MLCert is practical in that we extract learning procedures defined in Coq to executable code; thus procedures with proved generalization bounds can be trained and deployed in real systems. MLCert is well documented and open source; thus we expect it to be usable even by those without Coq expertise. To validate MLCert, which is compatible with external tools such as TensorFlow, we use it to prove generalization bounds on neural networks trained using TensorFlow on the extended MNIST data set.

1 Introduction

There is great optimism regarding the potential of artificial intelligence, and machine learning in particular, to automate tasks currently performed by humans. But there are also attendant challenges. In adversarial contexts, recent work at the intersection of machine learning and security has demonstrated that an attacker can exploit the sensitivity of a network to small input perturbations, so-called adversarial examples (Szegedy et al. 2013), in order to force misclassifications with high confidence. More broadly, machine learning systems can go wrong or be exploited in a variety of ways, at both training and inference time, as Papernot et al. comprehensively survey (Papernot et al. 2016). As machine learning becomes critical infrastructure in systems like autonomous vehicles, practitioners must ensure that machine-learned components do not invalidate high-level safety and security properties of the systems in which they are embedded.

In this paper, we make progress toward securing the foundations of machine learning practice by presenting a new system, MLCert, for building certified implementations of learning procedures, those with machine-checked generalization guarantees in a theorem prover. With respect to related work, which is primarily SMT-based (e.g., (Katz et al. 2017)), we target deeper functional specifications such as generalization error, or the expected error of a learned model when deployed on the distribution against which it was trained.

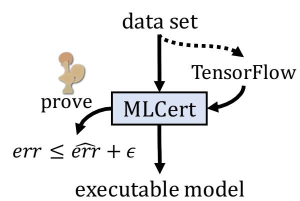

Figure 1: MLCert

Our technical approach (Figure 1) composes tools from interactive theorem proving, learning theory and statistics, and quantized neural networks (Hubara et al. 2016), those that learn low-precision weights. We train models either in MLCert, which is embedded in the Coq theorem prover (Bertot and Castéran 2013) or using an external tool such as TensorFlow (Abadi et al. 2016). We prove generalization bounds, relating expected to experimental error, using mechanized libraries of results from learning theory and statistics such as Chernoff inequalities, treating learning procedures as probabilistic programs with a denotational semantics in the style of (Kozen 1985). To validate experimental error of the Coq implementation of a model when run on a set of examples, we extract models to executable OCaml or Haskell code, which we then compile and run against an input data set. Statistical techniques for proving generalization are dependent on the complexity of the hypothesis class being learned. To tighten bounds without compromising performance, we use techniques from neural network quantization (Jacob et al. 2017) to learn accurate models with nontrivial generalization bounds for the extended MNIST (EMNIST) digit classification task (Cohen et al. 2017).

Contributions. To summarize, the primary technical contributions of this paper are as follows:

- We present MLCert (§§ 3 and 4), the first system for learning executable classifiers with certified generalization guarantees, bounding with mechanical proof in a theorem prover the expected error of machine-learned models.
- MLCert certifies generalization of the resulting classifiers, not of the training process, and thus is parametric in, and therefore compatible with, deep learning frameworks such as TensorFlow and PyTorch (Paszke et al. 2017). We demonstrate by proving generalization guarantees for quantized neural networks trained using TensorFlow (§ 5), for the EMNIST digit classification task (§ 6).
- Our implementation of MLCert is open source online.[1]

[1] `http://MLCert.org`

Limitations. The formal generalization guarantees we prove in this paper, that expected error on an underlying distribution D is close to training or test error on a sample S drawn from D, do not preclude adversaries that poison the training data, thus distorting our view of the underlying distribution D; or interfere with sampling, thus invalidating assumptions like independence; or shift the distribution from which examples are drawn during deployment (so-called distribution drift (Papernot et al. 2016)).

Because we use statistical techniques such as Chernoff bounds, our approach also requires that training and test data are large relative to model size, which is unrealistic for models like those deep neural networks that have many more parameters than training and test examples. Our current best generalization bounds from training error for EMNIST (Section 6) are still quite loose (77%). Our best bounds from test error on EMNIST are better (94%) but require many test examples ($40k$), which is unrealistic for some data sets. Nevertheless, by proving generalization bounds in Coq, we do get (probabilistic) guarantees – assuming sufficient training or test data – of the absence of overfitting errors. Our approach is also extensible to stochastic neural networks (Hinton and Van Camp 1993; McAllester 1999; Langford and Caruana 2002) for which researchers have recently achieved reasonably tight bounds even for large networks (Dziugaite and Roy 2017). We plan to apply MLCERT to stochastic nets in future work.

Why Prove Generalization? Machine learning practitioners ensure that a model generalizes by evaluating its performance on a holdout test set. Why should one care, then, whether a model is *proved* to generalize? By bounding expected error from training error, one no longer needs a test set, which can be useful – at least for very small model classes – when data is scarce. Using a test set also requires that one make additional independence assumptions (parameters are selected independently from the holdout set), which have been confounded by practices like *p-hacking* (Head et al. 2015). By certifying generalization bounds in Coq, we build proof artifacts that can be disseminated and checked with high assurance, facilitating replicable science.

From the perspective of the designer of high assurance software, generalization is useful in another way, as a natural specification of classifiers that increasingly form components of larger verified systems. For example, one might show, from bounded expected error of a neural network approximating a control law, that a brake controller will with high probability apply enough force to safely brake a car. Specifying and proving generalization bounds within a theorem prover facilitates the proof of end-to-end results such as these – the brake controller safely halts the car – with very high assurance. Certified generalization bounds may have application in scientific computing as well, to confirm with high assurance error bounds on, e.g., models for background noise classification in the Higgs boson LHC experiments (Baldi, Sadowski, and Whiteson 2014).

Trusted Computing Base. Our prototype currently assumes the following two textbook results as axioms: Pinsker's inequality, stating that the relative entropy of two distributions is bounded by a function of their total variation distance; and Gibb's inequality, stating that the relative entropy of two distributions is nonnegative. We axiomatize vector and floating point types and operations, as well as cardinality lemmas for vectors and floating point numbers. Our results also depend, as is typical of mechanized developments in the Coq theorem prover, on the correctness of tools in the Coq ecosystem such as the Coq proofchecker and Coq extraction.

2 Background

Theorem Proving. Theorem provers such as Coq (Bertot and Castéran 2013) enable programmers to build software and to prove its correctness all within the same programming environment. In this work, we use Coq for program proof – to validate with high assurance that implementations of learning procedures generalize – but also to manage the details of proofs of mathematical theorems, such as Hoeffding's inequality, upon which the software correctness arguments depend. Proofs in Coq proceed interactively: the programmer constructs a correctness argument in dialog with the proof management system, which displays a representation of the proof state at each point. Once the programmer completes a proof in Coq, it is checked by Coq's proof checker, a small kernel implementing Coq's internal dependent type theory. Correctness arguments in Coq therefore have small trusted computing bases: their correctness depends on the correctness of the Coq proof checker (and any assumed axioms) but not on that of the rest of the proof management system.

Learning Theory. The goal of supervised learning is to find a hypothesis $h \in H : X \to Y$, for some class H mapping X's to labels Y, that minimizes a metric like expected error $E(h(x), y)$ (or equivalently, maximizes a metric like expected accuracy $A(h(x), y)$) when presented with a previously unseen example (x, y) drawn from a distribution D. Because D is unknown, h is learned from a training set $T = [(x_1, y_1), \ldots, (x_m, y_m)]$ of examples, typically assumed to be drawn iid from D.

A hypothesis h has low *generalization error* if its expected accuracy $\mathsf{E}_{(x,y)\sim D}\left[A(h(x), y)\right]$ is not too far from its average empirical accuracy on training set T:

Definition 1 (Generalization Error)**.**

$$\left| \mathsf{E}_{(x,y)\sim D}\left[A(h(x), y)\right] - \frac{1}{|T|} \sum_{(x,y)\in T} A(h(x), y) \right|$$

If the generalization error of a hypothesis h is high, then h has *overfit* to the training set T, which can occur when the size of the hypothesis class H is large (or infinite) relative to the size of the training set T. Likewise, if H is small relative to the number of training examples, it is unlikely that a hypothesis $h \in H$ will overfit to T, an intuition that is formalized in a broad category of results called *Chernoff* bounds.

As a reminder of one commonly used Chernoff bound, called Hoeffding's inequality, consider m iid random variables $X_1, \ldots, X_m$ in the range $[0, 1]$ such that each X_i computes, for instance, the accuracy of a hypothesis h on sample (x_i, y_i) drawn from distribution D. Hoeffding's inequality states that the expected value of X_i (indeed, of all the X_i's – the random variables are identically distributed) is whp not

Listing 1: Statement of Corollary 1 in Coq

Theorem chernoff : $\forall(\epsilon{:}\mathbf{R})$(eps_gt0:$0{<}\epsilon$),
probOfR (prodR m D) [pred T : training_set |
let p := learn T **in** expVal D Acc p + ϵ < empVal Acc T p]
$\leq$ |Params| $*$ exp $(-2 * \epsilon^2 * m)$.

too much smaller than the empirical average of $X_1, \ldots, X_m$, assuming m is large enough. More precisely:

Theorem 1 (Hoeffding's Inequality). *Given m iid random variables $X_i \in [0,1]$ over D and $\epsilon \in (0, 1-E[X])$,*

$$\Pr\left[\mathsf{E}\,[X] + \epsilon < \frac{1}{m}\sum_i X_i\right] \leq e^{-2\epsilon^2 m}$$

For any fixed hypothesis h chosen independently of a data set T, Theorem 1 gives a bound on h's generalization error that decreases exponentially in the number m of examples (let X_i equal h's accuracy on example (x_i, y_i)).[2]

To bound the probability that *any* hypothesis $h \in H$ has high generalization error, including ones learned from T, one can combine Theorem 1 with a union bound to prove:

Corollary 1. *Given a training set T of m samples (x_i, y_i) drawn iid from D and any hypothesis h (which may be learned from T),*

$$\Pr\left[\mathsf{E}\,[A(h)] + \epsilon < \frac{1}{m}\sum_i A(h(x_i), y_i))\right] \leq |H| e^{-2\epsilon^2 m}$$

$\mathsf{E}\,[A(h)]$ is shorthand for expected accuracy of h on an example–label pair (x, y). The right-hand side of the bound is small if the number of examples m is large relative to $|H|$.[3] Likewise, the probability that expected accuracy is more than ϵ less than empirical accuracy grows smaller as ϵ increases.

Learning Theory in Coq. To make use of Theorem 1 and Corollary 1 in proofs about software, we must first translate into Coq. Listing 1 gives the Coq statement, for example, of Corollary 1. The term expVal D Acc p defines the expected accuracy of hypothesis p on a sample drawn from D, with:

Def Acc (p:Params) (i:$[0,m)$) (xy : $X{*}Y$) : $\mathbf{R}$:=
let (x,y) := xy **in if** predict p x == y **then** 1 **else** 0.

mapping parameters p and examples (x, y) to 1 or 0 depending on whether predict p x equals y. The probability on the left-hand side of Corollary 1 is computed by the Coq function probOfR, which returns the probability, with respect to the product distribution prodR m D of m example–label pairs drawn from D, of the predicate beginning [pred T | ...].

[2] A two-sided bound on absolute error yields right-hand side 2exp$(-2\epsilon^2 m)$. We prove this bound in Coq but do not use it here.

[3] Corollary 1 assumes H is finite but can be applied to infinite H using VC-dimension or Rademacher complexity theory, though we have not yet done so in this work. In execution settings, $|H| < \infty$ is nearly always valid (except in symbolic algebra, implementations use fixed-width floating-point numbers). VC-dimension theory or Rademacher complexity could, however, yield tighter bounds for some hypothesis classes.

Listing 2: Generic Learners

Record Learner (X Y Hypers Params : Type) : Type :=
mkLearner {
predict : Hypers $\rightarrow$ Params $\rightarrow X \rightarrow Y$;
update : Hypers $\rightarrow X \times Y \rightarrow$ Params $\rightarrow$ Params }.

The function predict : Params $\rightarrow X \rightarrow Y$ is a parameter (we instantiate it later, in § 3 and § 4, to particular prediction functions). The argument $i : [0, m)$ turns Acc into a family of functions, one for each of the m samples. Term empVal Acc T p gives the empirical accuracy of p on training set T. The hypothesis p is the result of a function learn mapping training sets to parameters. We elide two hypotheses in Listing 1: (1) the random variables defined by Acc are mutually independent for all p, which follows from mutual independence of the m samples, and (2) expected accuracy is strictly between 0 and 1.

3 MLCert By Example

Section 2 illustrated the translation of concentration bounds into mechanized form in Coq. Here we put the pieces together by applying these results to a case study (Perceptron).

We define learners generically (Listing 2) as pairs of functions: predict, which maps hyperparameters, parameters, and examples X to labels Y; and update, which maps hyperparameters, examples, and parameters to updated parameters of type Params. At this level of abstraction, the functions predict and update and their types are all generic.

Listing 3: Linear Threshold Classifiers

Variable n : nat. ($*$The dimensionality$*$)
X := float32_arr n. ($*$Examples: $n-$arrays of $32b$ floats$*$)
Y := bool. ($*$Boolean labels$*$)
Weights := float32_arr n. ($*$Parameters: weights, bias$*$)
Bias := float32.
Params : Type := Weights$\times$Bias.
Def predict (p : Params) (x : X) : Y :=
let (w, b) := p **in** f32_dot w x + b > 0.

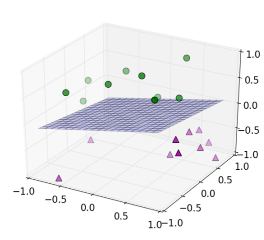

Figure 2: Model learned by the certified Perceptron of Listing 4

As an example of how one might instantiate the generic predict function of Listing 2, consider MLCert's definition of linear threshold classifiers (Listing 3). Such prediction rules are parameterized by a natural number n, the dimensionality of the example space. We define the example space X as size-n arrays of 32-bit floats and Y as bool. The parameter space is defined as Params := Weights$\times$Bias, the type of pairs of weights and biases. The prediction rule extracts a weight vector w and bias term b from p, then returns the result of evaluating $w \cdot x + b > 0$.

Listing 4: Perceptron Learner (Specializes Listing 2 and Listing 3)

```
Variable n : nat. (*The dimensionality*)
Hypers := { α:float32 }. (*Learning rate*)
Def update (h:Hypers) (xy:(X n)*Y) (p:Params n)
  : Params n :=
  let (x, y) := xy, (w, b) := p in
  if predict p x == y then p
  else (f32_map2 (λw_i x_i ⇒ w_i + α*y*x_i) w x, b + y).
Def PerceptronLearner
  : Learner (X n) Y Hypers (Params n) :=
  mkLearner (λ h:Hypers ⇒ predict) update.
```

Listing 4 defines a further specialization to Perceptron learning. The variable n : nat again indicates that the learner is parameterized by the dimensionality of the learning problem. The new type Hypers defines a record with a named field: α for learning rate. The function update defines how parameters p are updated when presented with a new example (x, y). Perceptron is error driven: if p correctly predicts x's label (using the generic linear threshold prediction rule of Listing 3) then update returns p unchanged. When predict p x mispredicts y, update returns the new weight vector in which each weight w_i equals $w_i + \alpha * y * x_i$ (MLCERT implicitly coerces the Boolean label y to $\{-1, 1\}$). The higher-order function f32_map2 produces a new vector component-wise from w and x according to the anonymous function beginning $\lambda w_i\ x_i.\ \ldots$. The overall result is a pair of the new weight vector (defined by f32_map) and the new bias term $(b + y)$.

To execute the Perceptron learner, we use Coq to automatically extract it to Haskell and then compile it against a small unverified shim (also in Haskell) that produces a training set. As illustration, consider the plot of Figure 2, which depicts the model learned by our MLCERT Perceptron on random linearly separable 3-dimensional data. To generate this plot, we instrumented our Haskell shim to print both the training set and model, which we then plotted using `matplotlib`.

4 Generalization Bounds

In Section 3, we outlined the use of MLCERT to build a simple Perceptron learner. In this section, we demonstrate the use of MLCERT to prove – with high assurance in Coq – that the implementations of learning procedures such as Perceptron generalize to unseen examples. The core of our proof strategy is to view learning procedures as probabilistic programs in the style of (Kozen 1985) and (Gordon et al. 2014), in which a program is understood mathematically, as in the denotational semantics of Figure 3, as the expected value of a function f over its results. To express bounds on the expected behavior of learning procedures, we use an observation command, observe, to filter only those executions that satisfy a particular postcondition, such as "model p fails to generalize". We express generalization results as bounds on the probability mass of these filtered executions (the probability of observing a nongeneralizing model should be small). Equivalently, one can think of our results as bounds on the expected value of the identity function in the subdistribution of program runs observed to produce models that fail to generalize. Our semantics, like that of (Gordon et al. 2014), is naturally unnormalized due to the presence of observe.

Listing 5: Generic Probabilistic Learners

```
Def main (D:X*Y→R) (m:nat) (ε:R) (init:Params) :=
  T ← sample m D; (*Draw m examples from D.*)
  p ← learn init T; (*Learn model p.*)
  observe (post D m ε) (p,T). (*Observe postcondition.*)
```

Listing 5 illustrates the general structure of probabilistic programs, main, that learn models from sampled training sets, and whose executions are then observed. The parameters of main are the underlying distribution D from which the training set is drawn, the number of training examples m, and the initial parameters init. The function first samples the training set T, then learns model p from init and T. The resulting model p and training set T are observed to satisfy a particular postcondition, post, which has the effect of filtering away those probabilistic executions that fail to satisfy post. In the context of MLCERT, we instantiate post to:

```
Def post D (m:nat) (ε:R) (pT:Params×training_set m) :=
  let (p,T) := pT in expVal D Acc p + ε < empVal Acc T p.
```

which, when specialized to sample size m, distribution D, and value ϵ, yields a postcondition expressing that p does not generalize well: the expected accuracy of p is more than ϵ lower than p's empirical accuracy on the training set T.

To demonstrate that the probability of seeing such an execution is low, we prove in Coq the following theorem:

```
Theorem main_bound : ∀D (m > 0) (ε > 0) (init:Params),
  main D m ε init (λ _ ⇒ 1) ≤ |Params| * exp (−2 * ε² * m).
```

stating that the subdistribution of executions that satisfy post, and therefore fail to generalize, has mass less than $|\mathsf{Params}| * \exp(-2 * \epsilon^2 * m)$, exactly the bound of Corollary 1. This theorem additionally assumes, as does Listing 1, the two hypotheses explained in the final paragraph of Section 2.

Holdouts. Theorem main_bound assumes that hypothesis p was learned from the same data set on which we evaluated empirical error. Assuming access to a holdout data set *not* used to learn p, we can apply MLCERT to prove tighter generalization bounds that scale, by Theorem 1, as $\exp(-2\epsilon^2 m)$ rather than $|\mathsf{Params}| \cdot \exp(-2\epsilon^2 m)$. To model the holdout protocol, we define in our Coq development a second probabilistic program, called main_holdout, which learns p on a sampled data set T_{train} but evaluates empirical error on an independently chosen T_{test}. We use the observe command to limit executions to those in which $\epsilon < 1 - \mathsf{expVal}\ D\ \mathsf{Acc}\ p$, a precondition of Theorem 1. We then prove a theorem in Coq just like main_bound but with main_holdout on the left-hand side and the tighter bound $\exp(-2 * \epsilon^2 * m)$ on the right.

Denotational Semantics. We give the denotational semantics that enables us to state **Theorem** main_bound in Figure 3. The upper part of the figure defines the syntax of commands c used in main, along with each command's type. The notation c : com A is read "c is a command with an outcome of type A". For example, sample(m, D) : com (training_set m) is a command, parameterized by a natural number m and distribution D, that has as outcome a training set of m examples.

Syntax and Type System $\boxed{\text{Commands } c : \mathsf{com}\ A}$

$$
\begin{array}{rcl}
\mathsf{sample}\ m\ D & : & \mathsf{com}\ (\mathsf{training\_set}\ m) \\
\mathsf{learn}\ \mathsf{h}\ \mathsf{e}\ \mathsf{init}\ T & : & \mathsf{com}\ \mathsf{Params} \\
\mathsf{observe}\ (P : \mathsf{pred}\ A)\ (a\!:\!A) & : & \mathsf{com}\ A \\
x \leftarrow (c_1 : \mathsf{com}\ A); & & \textit{composition} \\
(c_2 : A \to \mathsf{com}\ B) & : & \mathsf{com}\ B
\end{array}
$$

Denotational Semantics $\boxed{[\![c : \mathsf{com}\ A]\!](f : A \to \mathbf{R}) : \mathbf{R}}$

$$
\begin{array}{rcl}
[\![\mathsf{sample}\ m\ D]\!](f) & := & \sum_T (\mathsf{prodR}\ m\ D)(T) \cdot f(T) \\
[\![\mathsf{learn}\ \mathsf{h}\ \mathsf{e}\ \mathsf{init}\ T]\!](f) & := & f(\mathsf{learn'}\ \mathsf{h}\ \mathsf{e}\ \mathsf{init}\ T)) \\
[\![\mathsf{observe}\, P\, (a\!:\!A)]\!](f) & := & \mathsf{if}\ P(a)\ \mathsf{then}\ f(a)\ \mathsf{else}\ 0 \\
[\![x \leftarrow c_1; c_2]\!](f) & := & [\![c_1]\!](\lambda x. [\![c_2(x)]\!](f))
\end{array}
$$

Figure 3: Learners: Syntax and Probabilistic Semantics

Command learn, likewise, takes hyperparameters h, the number of epochs e, the initial parameters init, and training set T and has as outcome a learned model, of type Params. We discussed observe briefly in the previous section: It takes a predicate P over values of type A and values a of type A and filters out executions in which a does not satisfy P. The final command composes commands c_1 and c_2 in sequence, where c_1 has type com A and c_2 has type $A \to$ com B, notation for a function that takes an A as argument and returns a com B as result. The effect of sequencing is to first run c_1, resulting in an outcome x with which we instantiate and run c_2.

The lower part of the figure defines the denotational semantics of commands. Given a function f mapping A's outcomes to $\mathbf{R}$, the interpretation function $[\![c : \mathsf{com}\ A]\!](f)$ maps commands c to $\mathbf{R}$. Intuitively, f is a valuation function, the expected value of which we would like to compute in the distribution on outcomes generated by the interpretation of command c. As example, consider command sample(m, D). Its interpretation is $\sum_T (\mathsf{prodR}\ m\ D)(T) \cdot f(T)$, exactly f's expected value over the product distribution of m examples drawn from D. In the case of sample, f has type training_set $m \to \mathbf{R}$ (function from training sets to $\mathbf{R}$). However, f's type may differ from command to command.

As a second example, consider the interpretation of command $[\![\mathsf{observe}\ P\ a]\!](f)$, defined as $f(a)$ when a satisfies predicate P and 0 otherwise. The effect is to remove from the support of the valuation function f all those values a that fail to satisfy P. In the general case, in which a is produced by some computation c as in the program fragment $a \leftarrow c$; observe a P, the result is to set to 0 the mass of all executions of c producing outcomes that do not satisfy P.

The two commands whose interpretations we have not yet discussed are learn and composition ($x \leftarrow c_1; c_2$). We implement learn by applying f to the result of an auxiliary function learn′, which is defined as:

```
Def learn′ h e (init:Params) (T:training_set m) : Params :=
  fold (λ epoch p_epoch ⇒ (*for each epoch in [0, e):*)
    fold (λ xy p ⇒ (*for each xy in T:*)
      update h xy p)) (*update parameters p*)
   p_epoch T) init (enum [0, e)).
```

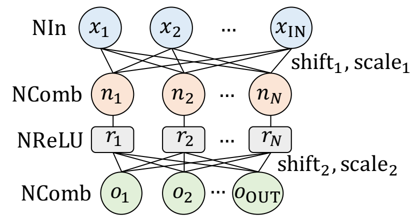

Figure 4: Structure of a network elaborated from Listing 7

Listing 6: Coq ReLU Networks

```
Inductive net : Type :=
| NIn : input_var → net (*A net is (1) an input_var*)
| NReLU : net → net (*or (2) a ReLU activation node*)
| NComb : list (param_var×net) → net.
 (*or (3) a linear combination of nets.*)
```

The auxiliary function is implemented in the context of a learning procedure, as in Listing 2, that defines the function update. It iterates over an outer and an inner loop (expressed as functional folds), the outer of which performs the inner loop e times, where e is the number of epochs. The inner loop repeatedly calls update on each xy example in the training set, producing new parameters p in each iteration. We interpret command composition $x \leftarrow c_1; c_2$ as the function composition of the interpretations of c_1 and c_2, namely: $[\![c_1]\!](\lambda x. [\![c_2(x)]\!](f))$. In other words, we first interpret c_1, then pass its outcome, called x, to $c_2(x)$, which is then interpreted with respect to the overall evaluation function f.

5 From TensorFlow To Coq

In this section we describe our representation of neural network architectures in Coq as well as our workflow for importing models trained using external tools like TensorFlow.

For logical consistency, functions in Coq must be proved terminating. To define network evaluation as a structurally recursive (and therefore terminating) function in Coq, we represent neural networks as (forests of) well-founded trees, as defined by type net in Listing 6. This datatype definition, built using Coq's user-defined inductive datatypes mechanism, says that a neural network (type net) is defined recursively as either (1) an input (input_var), (2) a ReLU activation function applied to a net, or (3) a linear combination of nets and weights, represented as a list of parameter weights (param_var) paired with nets.

As an example, consider the tree in Figure 4 rooted at node o_1. We represent this tree as the following net:

x_1:= NIn i_1; . . .; x_{IN}:= NIn i_{IN}
n_1:= NComb [($p_{\{1,1\}}$, x_1); ($p_{\{1,2\}}$, x_2); . . .; ($p_{\{1,\mathrm{IN}\}}$, x_{IN})], . . .
n_N:= NComb [($p_{\{N,1\}}$, x_1); ($p_{\{N,2\}}$, x_2); . . .; ($p_{\{N,\mathrm{IN}\}}$, x_{IN})]
r_1:= NReLU n_1, . . ., r_N:= NReLU n_N
o_1:= NComb [(p_1,r_1); (p_2,r_2); . . .; (p_N,r_N)]

in which p variables are parameters and i variables define inputs (neither of which are shown explicitly in Figure 4). A net with multiple outputs is a forest of inductive nets (with

Listing 7: Coq Network Kernels: One Hidden Layer. Kernels are parametric in the type T of weight parameters, the type S of shift/scale parameters, the number of hidden nodes N, the dimensionality IN of the input space, and the number OUT of network outputs.

```
Def Layer1Payload := AxVec IN T.
Def Layer1 := AxVec N Layer1Payload.
Def Layer2Payload := AxVec N T.
Def Layer2 := AxVec OUT Layer2Payload.
Record kernel:=
  { ss1:S×S; ss2:S×S; layer1:Layer1; layer2:Layer2 }.
```

sharing of nodes so the forest forms a DAG). For example, the complete network of Figure 4 is represented as the forest containing as roots the nodes $o_1, o_2, \ldots, o_{\mathsf{OUT}}$. This network representation, while specialized to ReLU nodes, could be generalized to support other kinds of activation functions.

Kernels. The right-hand side of main_bound (§ 4) scales in the cardinality of the parameter space. To represent the parameter spaces of neural networks compactly, while also facilitating cardinality proofs, we use a data structure called a *kernel* (Listing 7). A kernel completely determines a function in the parameter space (the space itself being fixed by the network architecture), and can be automatically elaborated to a DAG like the one in Figure 4 for execution from Coq. The kernels of Listing 7 are specialized to fully connected networks with one N-node hidden layer (Layer1). Layer2 defines the network's outputs, an OUT-length vector in which the entries, of type Layer2Payload, are N-vectors of weights of type T, one per hidden node. The type AxVec $size\ type$ defines axiomatized $size$-vectors containing values of type $type$. To import an external model to Coq, we use a Python script to generate a Coq source file containing a kernel, which can then be reasoned about in Coq and automatically extracted to OCaml or Haskell for execution.

To reduce the size of parameter spaces at the cost of a slight decrease in accuracy, we support quantization in kernels by associating with each layer a pair of *shift* and *scale* values, called ss1 and ss2 in Listing 7 for layers 1 and 2 respectively. During elaboration to an executable net, networks with low precision weights are converted to a higher precision format, then shifted and scaled by the values for each layer. This transformation enables low precision weights to take on a wide range of possible values. The sharing of shift and scale values among weights in a layer prevents blowup in the cardinality of the parameter space while still allowing sufficient representation flexibility. If the model being imported does not use quantized weights, we set default values of 0 and 1 for shift and scale respectively, to nullify their effect.

Generalization. We prove generalization of kernels in much the same way we proved generalization of Perceptron in Section 3, by bounding the mass of a probabilistic program that samples a training set, learns a network, then observes executions in which the learned classifier fails to generalize:

```
Variable oracle : ∀m:nat, training_set m → Params.
Def oracular_main (D:X∗Y→ R) (m:nat) (ϵ:R) :=
  T ← sample m D; (∗Draw m examples from D.∗)
  p ← ret (oracle m T); (∗Return external model p.∗)
  observe (post D m ϵ) (p,T). (∗Observe postcondition.∗)
```

The primary difference to Listing 5 is that we now assume an external learning procedure, **Variable** oracle, that produces models (type Param) from training sets, modeling the training of neural networks in an external tool like TensorFlow.

Because our generalization theorems result from statistical properties of the training set and hypothesis space, we prove generalization bounds even of external training procedures, such as oracle, about which we make no assumptions.

```
Theorem oracular_main_bound : ∀D (m > 0) (ϵ > 0),
  oracular_main D m ϵ (λ_⇒1) ≤ |Params|∗exp(−2∗ϵ²∗m).
```

This bound has the same right-hand side as that of Section 4 (main_bound); only the left-hand side is updated to include the oracular version of main. Our Coq development proves a second theorem, oracular_main_holdout_bound, limiting generalization error to the tighter $\exp(-2*\epsilon^2*m)$ when empirical error is evaluated on an independent test set. Both bounds additionally assume the two hypotheses of Listing 1, as elucidated at the end of Section 2.

6 Experiments

The generalization bounds of Sections 4 and 5 are most useful when they are reasonably tight, within a few percentage points of test error. In this section, we evaluate tightness by using TensorFlow to train two models on the EMNIST data set (Cohen et al. 2017), which we then import into MLCERT. In our experiments, we used a fully connected network architecture with a single hidden layer consisting of 10 ReLU units. We trained on a dataset of $240,000$ examples (EMNIST's training and validation sets) with $40,000$ examples set aside as holdout (EMNIST's test set). The holdout procedure, in which hypotheses are evaluated on an independent test set, yields reasonably tight bounds ($\sim 1.5\%$) while the union-bound procedure, in which empirical error is evaluated as in oracular_main on the same data set on which a hypothesis was trained, yields much looser bounds, from $\sim 15-43\%$. Our best union bound results required quantization to 2-bit weights, drastically decreasing the size of the parameter space but slightly decreasing accuracy.

To train quantized models, we inserted *fake quantization* operations from `tf.contrib.quantize` into the network computation graph, which simulated the quantization of weights while still allowing full 32-bit precision for effective numerical optimization. After training, we serialized the model weights to disk. Once a model was trained and stored, the weights were loaded by a Python script that generated a Coq source file containing the definition of a network kernel, as in Figure 7. The kernel was then elaborated to a forest in Coq, as in Figure 6, then extracted to executable code. To evaluate the accuracy of each model, we used a small unverified shim to load and feed batches of examples to the network, and to print the number of correct predictions in each batch. Then we used a Python script to compute the average number of correct predictions across all batches.

Results. Table 1 lists the accuracy of, and generalization bounds proved for, the two neural networks we trained using

Table 1: Experiments

W	**Params.** *(Bits)*	**Train**	**Test**	**Bound(ϵ)** Union	Holdout
2	15,944	0.925	0.926	77%(0.152)	91%(0.016)
16	127,104	0.958	0.957	53%(0.429)	94%(0.016)

TensorFlow on the EMNIST data set. W is the quantization scheme: either 2-bit quantized weights or 16-bit floating point with no quantization. "Params." is the size in bits of the parameter space for each network ($\log_2 |H|$). "Train" and "Test" are training- and test-set accuracy respectively. In "Bound(ϵ)", we report ϵ at confidence $1-10^{-9}$. To calculate "Union" bounds, we subtract ϵ from training error. To calculate "Holdout" bounds, we substract ϵ from test error.

In the "Union" case, the 2-bit quantized network has the tighter generalization bound, at $0.925 - 0.152(=\epsilon) = 77.3\%$ expected accuracy. While the 16-bit network achieved higher training-set accuracy (0.958), its generalization bound was looser, though still nontrivial, at 53%. Neither bound is very tight (for both networks, accuracy on a holdout set of $40,000$ examples is within a percentage point of training set accuracy). Nevertheless, there are optimizations we have yet to perform, all of which could further tighten bounds. One is to use a sparse representation of the kernel of Figure 7 (that is, only implicitly represent the 0 weights, and use regularization methods like those from (Srinivas, Subramanya, and Babu 2016) to encourage 0 weights during training). Another is to prove PAC-Bayes bounds for stochastic networks (McAllester 1999), using techniques from (Dziugaite and Roy 2017) to explicitly optimize bounds during training. This would require additional work in Coq to implement and prove the correctness of sampling from stochastic nets and to extend our mechanized Chernoff bounds to PAC-Bayes.

In the "Holdout" case, we get much tighter bounds, from 91% for the 2-bit quantized network to 94% for the 16-bit network. The value of $\epsilon = 0.016$ is the same for both networks because the holdout bound of (our mechanization of) Theorem 1 does not depend on model size. These bounds could likely be improved to close to 1.6% of the state-of-the-art test error on EMNIST by translating into Coq a state-of-the-art network for MNIST (e.g., (Wan et al. 2013)).

From Networks to Proofs. For the networks in Table 1, our Python script automatically generates statements and proofs, as corollaries of theorems oracular_main_bound and oracular_main_holdout_bound of Section 5, of generalization bounds specific to the network. For example, here is the "Union" bound we prove of the 2-bit network of Table 1:

```
Theorem tf_main_bound (ε > 0) init :
  tf_main D m ε init (λ _ ⇒ 1) ≤
  2^(4*16 + 784*10*2 + 10*10*2) * exp (−2 * ε^2 * m).
```

The number 4∗16+784∗10∗2+10∗10∗2 ($15,944$) is the size of the parameter space in bits. The sample size m is $240,000$. Function tf_main specializes oracular_main of Section 5 to the 2-bit quantized kernel we learned using TensorFlow. As when we first presented our generic generalization bound (Listing 1), we elide two assumptions: mutual independence and the bound on the range of expected accuracy.

7 Related Work

Recent results in ML verification already span a number of points in the design space. We survey the most relevant here.

Machine learning in interactive theorem provers. (Selsam, Liang, and Dill 2017) uses the Lean interactive theorem prover (de Moura et al. 2015) to formally verify the correctness of programs (e.g., Auto-Encoding Variational Bayes (Kingma and Welling 2013)) for optimized sampling of stochastic computation graphs. Rather than extracting such programs to a compilable representation, as we do in MLCert, (Selsam, Liang, and Dill 2017) use Lean's symbolic execution engine to interpret models interactively, executing numerically intensive tensor operations by calling out to an unverified high-performance library. (Selsam, Liang, and Dill 2017) also focus on only one particular model (stochastic computation graphs) and one particular specification (that backpropagation over computation graphs correctly computes gradients) while MLCert is more general, supporting arbitrary parameter spaces and learning algorithms, and integration of external tools such as TensorFlow. On the other hand, MLCert verifies that learning procedures generalize, not that each iteration of a learning procedure correctly computes gradients. Thus it is possible for an MLCert procedure to, e.g., fail to quickly converge to low training error, while still having a tight generalization bound.

Stability analysis of probabilistic programs. We prove generalization bounds in this work by appeal to statistical results such as Chernoff bounds, which do not depend on the details of implementations of learning procedures. An alternative approach by (Barthe et al. 2017) bounds generalization error by proving that learning procedures are stable (on only slightly different training sets, they produce only slightly different models). Because the (Barthe et al. 2017) approach reasons about properties of implementations of learning procedures, it is less easily integrated than MLCert with external tools such as TensorFlow (one would have to prove stability of core TensorFlow libraries, including optimizations, a daunting task). On the other hand, proofs from stability yield bounds that are independent of the size of the parameter space, and could therefore sometimes be tighter than those achievable using Chernoff inequalities.

Automated verification of neural network robustness. Recent work on adversarial examples (e.g., (Szegedy et al. 2013)) has spurred a series of results (e.g., (Katz et al. 2017; Huang et al. 2017; Kolter and Wong 2017)) in automated analysis of the *robustness* of neural networks: how much do model outputs differ when input examples are perturbed? Although robustness is distinct from algorithmic stability (the former applies to learned models while the latter applies to the learning process), robustness, like stability, is also deeply tied to generalization (cf. (Xu and Mannor 2012)). One goal of future work is to investigate whether the generalization bounds we prove in MLCert can be applied to do analysis and proof of neural network (expected) robustness.

8 Conclusion

This paper describes MLCERT, the first system for building executable machine-learned models with generalization guarantees certified in a theorem prover. MLCERT is compatible with externally trained tools such as TensorFlow, which we use to prove nontrivial generalization guarantees for quantized neural networks trained on EMNIST.

Acknowledgments

We thank Razvan Bunescu, Cindy Marling, Anindya Banerjee, and the AAAI anonymous reviewers for providing helpful comments on an earlier draft of this paper. We also thank Samuel Merten, who mechanized a lemma, and William Kanieski, who provided a paper-and-pencil proof. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1657358 and by the Ohio Federal Research Network (OFRN). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or of OFRN.

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, 265–283.

Baldi, P.; Sadowski, P.; and Whiteson, D. 2014. Searching for exotic particles in high-energy physics with deep learning. *Nature communications* 5:4308.

Barthe, G.; Espitau, T.; Grégoire, B.; Hsu, J.; and Strub, P.-Y. 2017. Proving expected sensitivity of probabilistic programs. *Proceedings of the ACM on Programming Languages* 2(POPL):57.

Bertot, Y., and Castéran, P. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

Cohen, G.; Afshar, S.; Tapson, J.; and van Schaik, A. 2017. EMNIST: an extension of MNIST to handwritten letters. *arXiv preprint arXiv:1702.05373*.

de Moura, L.; Kong, S.; Avigad, J.; Van Doorn, F.; and von Raumer, J. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, 378–388. Springer.

Dziugaite, G. K., and Roy, D. M. 2017. Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data. *arXiv preprint arXiv:1703.11008*.

Gordon, A. D.; Henzinger, T. A.; Nori, A. V.; and Rajamani, S. K. 2014. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, 167–181. New York, NY, USA: ACM.

Head, M. L.; Holman, L.; Lanfear, R.; Kahn, A. T.; and Jennions, M. D. 2015. The extent and consequences of p-hacking in science. *PLoS biology* 13(3):e1002106.

Hinton, G. E., and Van Camp, D. 1993. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual conference on Computational Learning Theory*, 5–13. ACM.

Huang, X.; Kwiatkowska, M.; Wang, S.; and Wu, M. 2017. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, 3–29. Springer.

Hubara, I.; Courbariaux, M.; Soudry, D.; El-Yaniv, R.; and Bengio, Y. 2016. Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR* abs/1609.07061.

Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M.; Howard, A.; Adam, H.; and Kalenichenko, D. 2017. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *arXiv preprint arXiv:1712.05877*.

Katz, G.; Barrett, C.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, 97–117. Springer.

Kingma, D. P., and Welling, M. 2013. Auto-Encoding Variational Bayes. *arXiv preprint arXiv:1312.6114*.

Kolter, J. Z., and Wong, E. 2017. Provable defenses against adversarial examples via the convex outer adversarial polytope. *arXiv preprint arXiv:1711.00851*.

Kozen, D. 1985. A probabilistic pdl. *Journal of Computer and System Sciences* 30(2):162–178.

Langford, J., and Caruana, R. 2002. (not) bounding the true error. In *Advances in Neural Information Processing Systems*, 809–816.

McAllester, D. A. 1999. PAC-Bayesian model averaging. In *Proceedings of the twelfth annual conference on Computational Learning Theory*, 164–170. ACM.

Papernot, N.; McDaniel, P.; Sinha, A.; and Wellman, M. 2016. Towards the science of security and privacy in machine learning. *arXiv preprint arXiv:1611.03814*.

Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A. 2017. Automatic differentiation in PyTorch.

Selsam, D.; Liang, P.; and Dill, D. L. 2017. Developing bug-free machine learning systems with formal mathematics. In Precup, D., and Teh, Y. W., eds., *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, 3047–3056. International Convention Centre, Sydney, Australia: PMLR.

Srinivas, S.; Subramanya, A.; and Babu, R. V. 2016. Training sparse neural networks. *CoRR* abs/1611.06694.

Szegedy, C.; Zaremba, W.; Sutskever, I.; Bruna, J.; Erhan, D.; Goodfellow, I. J.; and Fergus, R. 2013. Intriguing properties of neural networks. *CoRR* abs/1312.6199.

Wan, L.; Zeiler, M.; Zhang, S.; Le Cun, Y.; and Fergus, R. 2013. Regularization of neural networks using DropConnect. In *International Conference on Machine Learning*, 1058–1066.

Xu, H., and Mannor, S. 2012. Robustness and generalization. *Machine learning* 86(3):391–423.