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

Alexander Bagnall  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.

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 and can be downloaded as a supplement to this submission. http://MLCert.org

Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.
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 4) 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 near-term 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 makes it possible to prove 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 proof checker 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 $h \in H : X \rightarrow Y$, for some class $H$, that minimizes a metric like expected error $E(h(x), y)$ (or equivalently to maximize 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 sampled examples, typically assumed to be iid.

A hypothesis $h$ has low generalization error if its expected accuracy $E_{(x,y)\sim D}[A(h(x), y)]$ is not too far from its average empirical accuracy on training set $T$.

Definition 1 (Generalization Error).

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

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
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 \( m \) random variables \( X_i \in [0, 1] \) drawn iid from \( D \) and \( \epsilon \in (0, 1 - E[X]) \).

\[
\Pr \left[ \left| \frac{1}{m} \sum_{i=1}^{m} X_i - E[X] \right| > \epsilon \right] \leq 2e^{-2\epsilon^2 m}
\]

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 \( X_i = h(x_i, y_i) \)).

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 \( T \)),

\[
\Pr \left[ \left| \frac{1}{m} \sum_{i=1}^{m} A(h(x_i, y_i)) - E[A(h)] \right| > \epsilon \right] \leq 2e^{-2\epsilon^2 m}
\]

\( E[A(h)] \) 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 \( |H| \).

Likewise, the probability that expected accuracy is more than \( \epsilon \) less than empirical accuracy grows smaller as \( \epsilon \) increases.

**Learning Theory in Coq.** To make use of Theorem 1 and Corollary 1 in proofs about software, we must first translate into Coq for example. Listing 1 gives the Coq statement of the corollary. The term \( \text{expVal} \) defined in the corollary statement.

**Listing 1: Statement of Corollary 1 in Coq**

```coq
Theorem chernoff : \forall (\epsilon:R)(eps_gt0:0<\epsilon),
\[ \text{probOfR} \prod D T \text{ training_set} \]
let p := learn T \expVal D Acc p + \epsilon < \expVal Acc T p \]
\leq |Params| \ast \exp (-2 \ast \epsilon^2 \ast m).
```

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.

As an example of how one might instantiate the generic predict function of Listing 2, consider ML-Cert’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 \( \text{Params} := \text{Weights} \times \text{Bias} \), 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 3** defines a further specialization to Perceptron learning. The variable \( n : \mathbb{N} \) again indicates that the learner is parameterized by the dimensionality of the learning problem. The new type \( \text{Hypers} \) defines a record with a named field: \( \alpha \) 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 2), then update returns \( p \) unchanged. When predict \( p \) \( x \)
that the implementations of learning procedures such as
vector (defined by $\mathbf{W} \cdot \mathbf{X} := \mathbf{Y}$)
of main
observe
programs,
executions.
tion results as bounds on the probability mass of the filtered
probabilistic executions that satisfy a particular postcondition,
observe
use an observation command,
learning procedures). To express probabilistic guarantees, we
training sets (inputs) to distributions over model parameters
proof strategy is to view learning procedures as probabilistic
simple Perceptron learner. In this section, we demonstrate
Section 3, we outlined the use of MLC
decorated
set and model, which we then plotted using
matplotlib
set and model, which we then plotted using
the model learned by our MLC
unverified shim (also in Haskell) that produces a training set.

$\lambda w \in \mathbb{R} \rightarrow \mathbb{R} \quad \text{and} \quad \text{f32}_\text{map} (\lambda \mathbf{w}, \mathbf{x} \mapsto w_i + \alpha \mathbf{y} \cdot \mathbf{x}_i) \mathbf{w} \mathbf{x}$.

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 3, which
depicts the model learned by our MLCPerceptron 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 (Gordon et al. 2014) mapping distributions over
training sets (inputs) to distributions over model parameters
(outputs). By giving a denotational semantics to such pro-
grams (as shown in figure 3), we can prove probabilistic
results about the generalization of the resulting learned pa-
rameters, where the probability is over the initial distribution
over training sets (and any internal randomness used by the
learning procedures). To express probabilistic guarantees, we
use an observation command, observe, to filter those prob-
abilistic executions that satisfy a particular postcondition,
such as “model p fails to generalize”. We express generaliza-
tion results as bounds on the probability mass of the filtered
executions.

Listing 3: Linear Threshold Classifiers

```
Variable n : nat. (*The dimensionality*)
X := float32_arr n. (*Examples: n ¬ arrays of 32 ¬ bit floats*)
Y := bool. (*Boolean labels*)
Weights := float32_arr n. (*Parameters: weights and bias*)
Bias := float32.

Params : Type := Weights × Bias.

Def predict (p : Params) (x : X) : Y :=
  let (b, := p in f32_dot w x + b > 0.
```

mispredicts $y$, update returns the new weight vector in which
each weight $w_i$ equals $w_i + \alpha \mathbf{y} \cdot \mathbf{x}_i$ (MLCert implicitly co-
erces the Boolean label $y$ to $\{0, 1\}$). The higher-order func-
tion f32_map2 produces a new vector component-wise from
$w$ and $x$ according to the anonymous function beginning
$\lambda w \in \mathbb{R} \rightarrow \mathbb{R} \quad \text{and} \quad \text{f32}_\text{map} (\lambda \mathbf{w}, \mathbf{x} \mapsto w_i + \alpha \mathbf{y} \cdot \mathbf{x}_i) \mathbf{w} \mathbf{x}$. The overall result is a pair of the new weight
vector (defined by f32_map2) 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 MLCPerceptron 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 (Gordon et al. 2014) mapping distributions over
training sets (inputs) to distributions over model parameters
(outputs). By giving a denotational semantics to such pro-
grams (as shown in figure 3), we can prove probabilistic
results about the generalization of the resulting learned pa-
rameters, where the probability is over the initial distribution
over training sets (and any internal randomness used by the
learning procedures). To express probabilistic guarantees, we
use an observation command, observe, to filter those prob-
abilistic executions that satisfy a particular postcondition,
such as “model p fails to generalize”. We express generaliza-
tion results as bounds on the probability mass of the filtered
executions.

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 e) (p, T). (*Observe postcondition.*)
```

```
Variable n : nat. (*The dimensionality*)
Hypers := { α : float32 }, { Learning rate }

Def update (h : Hypers)(x y : (X n) × Y)(p : Params) : Params :=
  let (y, b) := p in
    if predict p x == y then p else
      f32_map2 (λ w, xi ⇒ 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.
```

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

To demonstrate that the probability of seeing such an exec-
tion 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) ≤ \lvert Params \rvert \exp (-2 * \epsilon^2 * m)
**

stating that the subdistribution of executions that satisfy post,
and therefore fail to generalize, has mass less than $\lvert Params \rvert \exp (-2 * \epsilon^2 * m)$, exactly the bound of Corollary 1.

**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 $\lvert Params \rvert \exp (-2 * \epsilon^2 * m)$. To model the holdout
protocol, we define in our Coq development a second prob-
abilistic program, called main_hooldout, 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 $\epsilon < 1 − \exp Val D Acc p$, a
precondition of Theorem 1. We then prove a theorem in Coq
just like main_bound but with main_hooldout on the left-hand
side and the tighter bound $\exp (-2 * \epsilon^2 * m)$ on the right.

**Denotational Semantics.** We have thus far glossed over
the denotational semantics that enables us to view programs
like main as distribution transformers, about whose poste-
rive distributions we can make observations. We give this
denotational semantics in Figure 3.

The upper part of the figure defines the syntax of
commands $c$ used in main, along with each com-
mand’s type. The notation $c : \text{com}$ is read “$c$ is a
command with an outcome of type $\text{com}$. For example,
sample$(m, D) : \text{com}$ is a command, parameter-
ized by a natural number $m$ and distribution $D$, that has as
Syntax and Type System

Commands \(c : \text{com } A\)

- `sample m D` : \(\text{com } (\text{training_set } m)\)
- `learn h e init T` : \(\text{com Params}\)
- `observe (P : \text{pred } A) (a : A)` : \(\text{com } A\)
- `x \gets (c_1 : \text{com } A); \text{command composition}\`
- `(c_2 : A \rightarrow \text{com } B)` : \(\text{com } B\)

Denotational Semantics

\[\llbracket c : \text{com } A \rrbracket (f : A \rightarrow R) : R\]

- `\llbracket \text{sample } m D \rrbracket (f)` := \(\sum_{T} (\text{prodR } m D)(T) \cdot f(T)\)
- `\llbracket \text{learn } h e \text{ init } T \rrbracket (f)` := \(f(\text{learn}' h e \text{ init } T)\)
- `\llbracket \text{observe } P (a : A) \rrbracket (f)` := \(\text{if } P(a) \text{ then } f(a) \text{ else } 0\)
- `\llbracket x \gets c_1; c_2 \rrbracket (f)` := \([c_1]([\lambda x.[c_2](x)])(f)\)

Figure 3: Learners: Syntax and Probabilistic Semantics

outcome a training set of \(m\) examples. Command `learn', likewise, takes hyperparameters \(h\), the number of epochs \(e\), the initial parameters \(p\), and training set \(T\) and has as output 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 \rightarrow com B`, notation for a function that takes an \(A\) as argument and returns a \(B\) 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 random variable \(f\) mapping \(A\)'s outcomes to \(R\), the interpretation function \(\llbracket c : \text{com } A \rrbracket (f)\) maps commands \(c\) to \(R\). Intuitively, \(f\) is a valuation function, the expected value of which we would like to compute on the distribution on outcomes generated by command \(c\). As example, consider command `sample(m, D)`. Its interpretation is \(\sum_{T} (\text{prodR } m D)(T) \cdot f(T)\), 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 \rightarrow R` (function from training sets to \(R\)). However, \(f\)'s type may differ from command to command.

As a second example, consider the interpretation of command `observe P a 0 (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 \gets 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 \gets c_1; c_2`). We implement `learn` by applying `f` to the result of an auxiliary function `learn'`, which is defined as:

\[
\text{Def } \text{learn}' \ h \ (\text{init:Params}) (T \text{training_set } m) : \text{Params} :=
\begin{align*}
\text{fold } (\lambda \text{epoch } p_{\text{epoch}} \Rightarrow (*\text{for each epoch in } [0, e]*) \\
\text{fold } (\lambda x y p \Rightarrow (*\text{for each } x y \text{ in } T*:)) \\
\text{update } h \ x \ y \ p \text{)} (*\text{update parameters } p*) \\
p_{\text{epoch}, T} \text{ init (enum } [0, e]).
\end{align*}
\]

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 \(x y\) example in the training set, producing new parameters \(p\) in each iteration.

We interpret command composition `x \gets 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:

\[
x_1 := \text{Nn } i_1; \ldots; x_m := \text{Nn } i_m \\
\text{n}_1 := \text{NComb } \{(p_{(1,1)}, x_1); \{p_{(1,2)}, x_2\}; \ldots; \{p_{(1,n)}, x_m\}\} \\
\text{n}_2 := \text{NComb } \{(p_{(2,1)}, x_1); \{p_{(2,2)}, x_2\}; \ldots; \{p_{(2,n)}, x_m\}\} \\
\text{r}_1 := \text{NReLU } \text{n}_1; \ldots; \text{r}_n := \text{NReLU } \text{n}_N \\
\text{o}_1 := \text{NComb } \{(p_{1,1}; r_1); \{p_{2,2}; r_2\}; \ldots; \{p_{n,n}; r_n\}\}
\]

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
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 \( o_1, o_2, \ldots, o_{\text{OUT}} \). 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 5 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:

```coq
Variable oracle : \forall m:nat, training_set m \rightarrow Params.
Def oracular_main (D:X \rightarrow Y \times R) (m:nat) (\epsilon:R) :=
  T \leftarrow \text{sample } m D; (*\text{Draw } m \text{ examples from } D.*)
```

The primary difference to Listing 5 is that we now assume an external learning procedure, \texttt{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 : \forall D \times m > 0 \epsilon > 0, oracular\_main D m \epsilon (\lambda \Rightarrow 1) \leq |\text{Params}| \times \exp(-2 \epsilon^2 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.

### 6 Experiments

The generalization bounds of Sections 3 and 4 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 \textit{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 TensorFlow on the EMNIST data set. \( W \) is the quantization scheme: either 2-bit quantized weights or 16-bit floating point with no quantization. \textquote{Params} is the size in bits
of the parameter space for each network (log₂ |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 subtract ε from test error.

In the “Union” case, the 2-bit quantized network has the tighter generalization bound, at 0.925 – 0.152(ε) = 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 ε = 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 :

\[ tf\_main\ D\ m\ \epsilon\ \text{init} \Rightarrow (\lambda \_ \Rightarrow 1) \leq 2^{4*16 + 784*10 + 12 \cdot 10 + 10*2} \cdot \exp \left( -2 \cdot \epsilon^2 \cdot m \right). \]

The number \( 4*16 + 784*10 + 12 \cdot 10 + 10*2 \) (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 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), (Hu 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 guar-
anties certified in a vtheorem 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.

References


