

Abstract
Frank Rosenblatt invented the perceptron algorithm in 1957 as part of an early attempt to build “brain models”, artificial neural networks. In this paper, we apply tools from symbolic logic such as dependent type theory as implemented in Coq to build, and prove convergence of, one-layer perceptrons (specifically, we show that our Coq implementation converges to a binary classifier when trained on linearly separable datasets).

Our perceptron and proof are extensible, which we demonstrate by adapting our convergence proof to the averaged perceptron, a common variant of the basic perceptron algorithm. We perform experiments to evaluate the performance of our Coq perceptron vs. an arbitrary-precision C++ implementation and against a hybrid implementation in which separators learned in C++ are certified in Coq. We find that by carefully optimizing the extraction of our Coq perceptron, we can meet – and occasionally exceed – the performance of the arbitrary-precision C++ implementation. Our hybrid Coq certifier demonstrates an architecture for building high-assurance machine-learning systems that reuse existing codebases.

Verified Perceptron Convergence Theorem

Charlie Murphy
Princeton University, USA
tcm3@cs.princeton.edu

Patrick Gray  Gordon Stewart
Ohio University, USA
pg219709@ohio.edu/gstewart@ohio.edu

1. Introduction
Frank Rosenblatt developed the perceptron in 1957 (Rosenblatt 1957) as part of a broader program to “explain the psychological functioning of a brain in terms of known laws of physics and mathematics…” (Rosenblatt 1962, p. 3). To again quote (Rosenblatt 1962),

A perceptron is first and foremost a brain model, not an invention for pattern recognition… its utility is in enabling us to determine the physical conditions for the emergence of various psychological properties.

The classic story is that Minsky and Papert’s book Perceptrons (Minsky and Papert 1969) put a damper on initial enthusiasm for early neural network models such as Rosenblatt’s by proving, for example, that one-layer perceptrons were incapable of learning certain simple boolean functions. But as Minsky and Papert themselves put it in the prologue to the 1988 edition of their book, their early negative results were only part of the story:

Our version [of the history] is that the progress [on learning in network machines] had already come to a virtual halt because of the lack of adequate basic theories, and the lessons in this book provided the field with new momentum. . . .

Minsky and Papert go on to argue that the “symbolist” paradigm they advocated in those early years of artificial intelligence in the 1960s and 70s laid the groundwork for new ways of representing knowledge — that “connectionism” as embodied by the early network models of Rosenblatt and others (and even of Minsky himself1) was foundering not because Perceptrons drove researchers away, but because the early connectionist work lacked firm theoretical foundation.

In this paper, we step back from the historical debate on connectionism vs. symbolism to consider the perceptron algorithm in a fresh light: the language of dependent type theory as implemented in Coq (The Coq Development Team 2016). Our work is both new proof engineering, in the sense that we apply interactive theorem proving technology to an understudied problem space (convergence proofs for learning algorithms), but also reformulation of an existing body of knowledge, in the hope that our work increases the profile of classic machine-learning algorithms like the perceptron among programming languages researchers.

While the single-layer perceptron we formalize and for which we mechanically prove convergence bounds in this paper is less sophisticated than more modern machine learning methods – SVM or deep neural networks trained via backpropagation, for example – it’s nevertheless a useful algorithm, especially in its extensions to the averaged perceptron, which we have also implemented and proved convergent (Section 4.2), or to MIRA (Crammer and Singer 2003). By formalizing and proving perceptron convergence, we demonstrate a proof-of-concept architecture, using classic programming languages techniques like proof by refinement, by which further machine-learning algorithms with sufficiently developed metatheory can be implemented and verified. The hybrid certifier we describe in Section 5 additionally shows that high assurance is possible without sacrificing performance, and even re-using existing (potentially previously optimized) implementations.

Toward these ends, this paper makes the following specific contributions.

• We implement perceptron and averaged perceptron in Coq.
• We prove that our Coq perceptron implementations converge as long as their training data are linearly separable. Our proofs are constructive in two senses: First, we use vanilla Coq with no axioms. Second, under the assumption that some particular separating hyperplane is known, our termination proof calcu-

lates explicit bounds on the number of iterations required for perceptron to converge.

- We implement a hybrid certifier architecture, in which a C++
  perceptron oracle is used to generate separating hyperplanes
  which are then certified by a Coq validator.

- Our Coq perceptron is executable and reasonably efficient. We
  evaluate (Section 7) its performance, when extracted to Haskell,
  on a variety of real and randomly generated datasets against
  both a baseline C++ implementation using arbitrary-precision
  rational numbers and against the hybrid certifier architecture
  supplied with a floating-point C++ oracle. When extraction is
  optimized (Section 6), the performance of our Coq perceptron
  is comparable to – and sometimes better than – that of the C++
  arbitrary-precision rational implementation.

In support of these specific contributions, we first describe the
key ideas underlying the perceptron algorithm (Section 2) and
its convergence proof (Section 3). In Sections 4 and 5 we report
on our Coq implementation and convergence proof, and on the
hybrid certifier architecture. Sections 6 and 7 describe our extraction
procedure and present the results of our performance comparison
experiments. Sections 8 and 9 put our work in this paper in its
broader research context with respect to the interactive theorem
proving literature. The code and Coq proofs we describe are open
source under a permissive license and are available online.2

1.1 Limitations and Possible Extensions

Although our Coq perceptron implementation is verified convergent
(Section 4) and can be used to build classifiers for real datasets
(Section 7, 11), it is still only a proof-of-concept in a number of
important respects. For example:

**Single- vs. Multi-Layer.** Our convergence proof applies only
to single-node perceptrons. Multi-node (multi-layer) perceptrons
are generally trained using backpropagation. When a multi-layer
perceptron consists only of linear perceptron units (i.e., every
activation function other than the final output threshold is the
identity function), it has equivalent expressive power to a single-
ode perceptron. We suspect that parts of our implementation and
proofs could be expanded or re-used to prove termination in this
(admittedly somewhat limited) case.

**Linear vs. Logistic.** Perceptrons equipped with sigmoid rather
than linear threshold output functions essentially perform logistic
regression. Such perceptrons aren’t guaranteed to converge (Chang
and Abdel-Ghaffar 1992), which is why general multi-layer perceptrons
with sigmoid threshold functions may also fail to converge.
Our convergence proof (Section 4) does not generalize to this case.

**Partial vs. Total.** Our perceptron is guaranteed to terminate only
on linearly separable datasets (it’s a partial function with guaranteed
termination behavior for a subset of its input space). As we discuss
in Section 6 one direct extension is to construct and prove correct
decision procedure for linearly separability (using, for example, a
convex hull algorithm). We could then compose this decision
procedure with our perceptron to yield a total function over all
training sets, returning a proof of inseparability in cases where our
current perceptron just infinite loops.

2. **Perceptron**

The perceptron is a supervised learning algorithm that computes
a decision boundary between two classes of labeled data points.
There may be many such decision boundaries; the goal is to learn
a classifier that generalizes well to unseen data. The data points are
represented as feature vectors \(x \in \mathbb{R}^n\), where each feature is a

\[
\text{for } E \text{ epochs or until convergence do} \\
\text{for } (x, l) \in T \text{ do} \\
y = \text{sign}(w^T x) \\
\text{if } y \neq l \text{ then} \\
w = w + x_l
\]

**Figure 1: Perceptron algorithm**

numerical delineation of an attribute possibly correlated with the
given class label \(l \in \{-1, +1\}\). A learned weight model \(w \in \mathbb{R}^n\)
and a bias parameter \(w_0\) define, respectively, the orientation and
location of the boundary.

To predict class labels \(y\), the perceptron projects a given feature
vector \(x\) onto \(w\) and clamps the result as either positive or negative:

\[
y = \text{sign}(w^T x + w_0) = \begin{cases} 
1, & \text{if } w^T x + w_0 \geq 0 \\
-1, & \text{if } w^T x + w_0 < 0 
\end{cases} 
\]

**Figure 2: AC Bound**

Definition 1 (Linear Separability).

**Linearly Separable** \(T \triangleq \exists w^*, w_0^* \colon \forall (x, l) \in T. l = \text{sign}(w^{*T} x + w_0^*)\).

A data set is linearly separable if there exists a weight vector \(w^*\)
and bias term \(w_0^*\) such that all feature vectors \(x\) in the training
data \(T\) have predicted sign equal to their true class label \(l\).
3. Perceptron Converges, Informally

As far as we are aware, (Paper|1961) and then (Block|1962) were the first to prove that the perceptron procedure converges. Figure 2 gives intuition for the proof structure.

Assume \( k \) is the number of vectors misclassified by the perceptron procedure at some point during execution of the algorithm and let \( |w_k - w_0|^2 \) equal the square of the Euclidean norm of the weight vector (minus the initial weight vector \( w_0 \)) at that point. The convergence proof proceeds by first proving that \( |w_k - w_0|^2 \) is bounded above by a function \( Ck \), for some constant \( C \), and below by some function \( Ak^2 \), for some constant \( A \). (The constants \( C \) and \( A \) are derived from the training set \( T \), the initial weight vector \( w_0 \), and the assumed separator \( w^* \).) As the perceptron algorithm proceeds, the number of misclassifications \( k \) approaches \( C/A \). The overall result follows from this “AC” bound and the fact that, at each iteration of the outer loop of Figure 4 until convergence, the perceptron misclassifies at least one vector in the training set (sending \( k \) to at least \( k + 1 \)).

To derive \( A \), observe that \( w_k - w_0 \), the weight vector at step \( k \), can be rewritten in terms of \( w_{k-1} \) and the most recently misclassified element \( x_k \) and its label \( l_k \) as:

\[
\begin{align*}
    w_k &= w_{k-1} + x_k l_k \\
    &= w_0 + x_1 l_1 + \cdots + x_k l_k
\end{align*}
\]

(4)

Subtracting the initial weight vector \( w_0 \) and multiplying both sides by \( w^* \), the transpose of the assumed separating vector \( w^* \), results in

\[
  w^* (w_k - w_0) = w^* (x_1 l_1 + \cdots + x_k l_k)
\]

(5)

Let \( a = \min_{(x,l) \in T} w^* x l \) be the minimum vector product \( w^* x l \) across all vectors label pairs \((x,l)\) in the training set \( T \). Then

\[
  w^* (w_k - w_0) \geq w^* (x_1 l_1 + \cdots + x_k l_k)
\]

(6)

\[|w^*|^2 |w_k - w_0| \geq |w^* (w_k - w_0)|^2 \geq (ak)^2\] (8)

where \[3\] follows from \[7\] and the Cauchy-Schwarz inequality. From \[3\] it’s straightforward to derive that \( A = \frac{a^2}{w^*} \).

\[C \] is derived by a similar set of inequalities:

\[
  |w_k - w_0|^2 = |w_{k-1} + x_k - w_0|^2 = |w_{k-1} - w_0|^2 + 2(w_{k-1} - w_0) x_k + |x_k|^2
\]

(9)

where \[7\] follows by foiling the square of the Euclidean norm. Assuming \( \forall i \in [1,k], w^*_i x_i \leq 0 \), which holds after each \( x_i \) has been normalized by multiplying it by its class label, we get that

\[
|w_k - w_0|^2 \leq |w_{k-1} - w_0|^2 - 2(w_{k-1} - w_0) x_k + |x_k|^2 \]

(10)

\[
|w_k - w_0|^2 \leq |x_1|^2 + \cdots + |x_k|^2 - 2w^*_i (x_2 + \cdots + x_k)
\]

(11)

Inequality \[10\] follows from \[9\] and from nonpositivity. Summing over all \( i \) from 1 to \( k \) gives \[11\] from \[10\].

Define \( M = \max_{x \in T} |x|^2 \) and \( \mu = \min_{x \in T} w^*_i x \). Then from inequality \[11\] we have that

\[
  |w_k - w_0|^2 \leq Mk - 2\mu k = (M - 2\mu)k
\]

(12)

giving \( C = M - 2\mu \).

4. Implementation and Formal Proof

In our Coq perceptron, we make three small changes to the Section 2 algorithm:

- We use \( \mathbb{Q} \)-instead of real-valued vectors.
- In our representation of training data sets, we use \( \mathbb{bool} \) instead of \( \mathbb{Q} \) to record class labels \( +1, -1 \). Thus the type system enforces canonicity of the labels.
- We record the bias term \( w_0 \) by consing it to the front of the decision boundary \( w \). Thus our \( w \) vectors are of size \( 1 + \# \text{features} \).

Listing 1 gives the basic definitions used by our perceptron implementation and in the statement of the convergence theorem.

Listings 2 gives the main definitions.

Listing 2: Coq perceptron

Fixpoint inner_perceptron \( \{n : \mathbb{nat}\} \) \( \{E : \mathbb{nat}\} \) \( \{C : \mathbb{nat}\} \) \( \{T : \mathbb{list} \{\mathbb{Q}vec \mathbb{n} \mathbb{+} \mathbb{bool}\}\} \) \( \{w : \mathbb{Q}vec \mathbb{n} \mathbb{+} \mathbb{1}\} \) \( \{\mathbb{match} T \mathbb{with} \{\mathbb{nil} \Rightarrow \mathbb{None}\} \| \{x, l : \mathbb{T} \Rightarrow \}\) if \( \mathbb{correct \_class} \{i : \mathbb{Q}vec \mathbb{n} \mathbb{+} \mathbb{bool}\} \{l : \mathbb{bool}\} \Rightarrow \mathbb{true} \)
\( \mathbb{match} C \mathbb{with} \{\mathbb{0} \Rightarrow \mathbb{None}\} \| \{\mathbb{1} \Rightarrow \mathbb{Some} w\} \)
\( \mathbb{end} \)
\( \mathbb{end} \)
\( \mathbb{end} \)
\( \mathbb{end} \)
\( \mathbb{end} \)
\( \mathbb{end} \)
\( \mathbb{end} \)
\( \mathbb{end} \)
\( \mathbb{end} \)
\( \mathbb{end} \)
\( \mathbb{end} \)
\( \mathbb{end} \)

The fixpoint \( \mathbb{inner \_perceptron} \), which corresponds to the inner loop in Figure 4, does most of the work. Its recursion parameter, \( T \), is a list of training vectors paired with their labels. The function iterates through this list, checking whether each training vector is correctly classified by the current decision boundary \( w \). Upon correct classification of an \( x \), \( \mathbb{inner \_perceptron} \) simply moves to the next training vector. Upon misclassification, we let the new decision vector, \( w' \), equal the vector sum of \( w \) and \( x \) multiplied by its class label. The function then continues iterating through the remaining training samples \( T' \).

Listing 2’s second fixpoint, \( \mathbb{implement} \), implements the outer loop of Figure 4. Its recursion parameter is a natural number.

\( \mathbb{The \ function consb} \) (Listing 4) conses 1 to \( x \) to account for \( w \)’s bias term.
E, a value typically called “fuel” in interactive theorem proving that bounds the number of inner _perceptron_ epochs. In our formal convergence proof (Section 4.1), we show that for any linearly separable training set T, there exists an E large enough to make perceptron terminate with Some w’. By the definition of perceptron, convergence only occurs when the algorithm has settled on a w that correctly classified all the training vectors in T (inner _perceptron_ T w = None). Thus soundness (if perceptron converges, it does so with a vector w’ that separates the training set T) is trivial.

To state the convergence theorem, we first formalize (Listing 3) what it means for a data set T to be linearly separable. The binary predicate correctly _classifiedP_ T w states that vector w correctly classified training set T (a list of vector-label pairs). A data set T is linearly separable when there exists a w’ such that w” correctly classifies T. The main convergence result is thus:

**Theorem 1** (Perceptron Converges).

\[
\forall\{\text{n:nat}\} (T : \text{list (Qvec n+bool)}) (w_0 : \text{Qvec n+1}), \\
(\text{linearly separable } T \rightarrow \\
(\exists (w : \text{Qvec n+1}) (E_0 : \text{nat}), \forall E : \text{nat}, E \geq E_0 \rightarrow \\
\text{perceptron } E \ T \ w_0 = \text{Some } w) \land \\
(\forall E : \text{nat} : (w : \text{Qvec n+1}), \\
\text{perceptron } E \ T \ w_0 = \text{Some } w \rightarrow \text{correctly _classifiedP } T \ w).
\]

**Listing 3:** Linear Separability

**Definition** correctly _classifiedP_ {n : nat} :
list (Qvec n+bool) \rightarrow Qvec n+1 \rightarrow Prop ⊢
λ T w ⇒ List.Forall
(λ x l : (Qvec n+bool) ⇒ let (x, l) ≜ x in 
\text{correct class (Qvec dot w (consb x)) l = true}) T.

**Definition** linearly separable {n : nat} :
T : list (Qvec n+bool) : Prop ⊢
∃ w : Qvec n+1, correctly _classifiedP T w Lind.

For all training sets T and initial vectors w_0, T is linearly separable iff there is an E such that perceptron converges (to some separator w) when run on E_0 or greater fuel. Note that the second conjunct of this theorem is trivial, from the definition of the perceptron algorithm.

### 4.1 Formal Convergence Proof

What about the formal proof? By the definition of perceptron, the second conjunct (correctness) of Theorem 1 is easy. But as we outlined in Section 3 the first conjunct (convergence) requires a bit more work. In our exposition in this section, we break the proof into two major parts: **Part I** defines two alternative formulations of perceptron – to expose in the termination proof the feature vectors misclassified during each run – together with refinement proofs that relate the termination behaviors of the alternative perceptrons, while **Part II** composes a proof of the “AC” bound (Section 3) with the results from Part I to prove the overall Theorem 1. We consider each part in turn.

#### 4.1.1 Part I

The implementation of perceptron in Listing 2 returns only the final weight vector w’ (or None on no fuel) – it gives no indication of which training vectors were misclassified in the process. Yet the informal proof explicitly bounds the number of misclassifications. To get on handle these “misclassified elements” in our formal proof, we defined two alternative Fixpoints:

**Fixpoint** MCE {n:nat} (E:nat) T w : list (Qvec n+bool) ⊢ ...

**Fixpoint** perceptron_MCE {n:nat} (E:nat) T w

MCE returns a list of the vectors misclassified by the perceptron algorithm. The middle fixed point perceptron_MCE serves as a bridge between perceptron and MCE: it returns either None on no fuel or a pair of the final weight vector and the list of misclassified elements (as in MCE). We then prove the following lemmas to relate the convergence behavior of perceptron, perceptron_MCE, and MCE:

**Lemma** MCE_sub_perceptron_MCE :
\forall (n : \text{nat}) (w_0 : \text{Qvec n+1}) (T : \text{list (Qvec n+bool)}), \\
MCE E T w_0 = MCE E+1 T w_0 \rightarrow \\
\text{perceptron } MCE E+1 T w_0 = \text{Some } (MCE E T w_0).

**Lemma** perceptron_MCE_eq_perceptron :
\forall (n : \text{nat}) (T : \text{list (Qvec n+bool)}) (w_0 : \text{Qvec n+1}), \\
(\exists M, \text{perceptron } MCE E T w_0 = \text{Some } (M, w)) \leftrightarrow \\
\text{perceptron } E T w_0 = \text{Some } w.

The first lemma proves that MCE’s convergence behavior refines that of the second function perceptron_MCE. If at E fuel, MCE E T w_0 has reached a fixed point (the list of misclassified feature vectors is stable regardless of how much additional fuel we provide), then perceptron_MCE on E+1 fuel returns the same list of misclassified vectors, together with final weight vector equal the vector sum of w_0 and each vector in MCE E T w_0 multiplied by its class label.

The second lemma proves an equitermination property: the function perceptron_MCE converges to Some (M, w) on training set T iff perceptron also converges to w.

#### 4.1.2 Part II

The main difficulty in part II is proving the AC bound on the length of MCE E T w_0, the number of misclassified feature vectors. Note that the length of MCE E T w_0 need not be less than or equal to |T|, the size of the training set; It’s possible that the same feature vector is misclassified during multiple iterations of the perceptron outer loop.

Our formal statement of the MCE bounds lemma is:

**Lemma** MCE_Bounded :
\forall \{\text{n:nat}\} (T : \text{list (Qvec n+bool)}) (w_0 : \text{Qvec n+1}), \\
\text{linearly separable } T \rightarrow \\
\exists A, B, C : \text{nat}, A \neq 0 \land B \neq 0 \land C \neq 0 \land \\
\forall E : \text{nat} : A \ + \ |MCE E T w_0|^2 \leq B \ + \ \text{Qvec normsq \ (Qvec sum \ (MCE E T w_0))} \leq C \ + \ |MCE E T w_0|.

**Qvec normsq** takes the square of the Euclidean norm of its input vector, while **Qvec sum** computes the vector sum of all the vectors in the provided input list.

**Proof**. Our proof of the lower bound makes use of the Cauchy-Schwarz inequality:

**Lemma** Cauchy_Schwarz :
\forall \{\text{n:nat}\} (x_1 : x_2 : \text{Qvec n}), \\
\text{Qvec dot x_1} x_2 \ + \ \text{Qvec dot x_1} x_2 \leq \\
\text{Qvec normsq} x_1 \ + \ \text{Qvec normsq} x_2.

The upper bound is a bit easier; in particular, it does not require even that T is linearly separable. For further details, see the Coq development.

To prove the overall convergence result (Theorem 1), we use Lemma 1 (MCE Bounded).
Proof of Theorem[7] (Perceptron Convergence). Lemma[1] and the following arithmetic fact
\( \forall xyz : \mathbb{N}. x \neq 0 \land y \neq 0 \land z > y/x \Rightarrow x z^2 > y z \)
together imply that the maximum length of MCE \( T w_0 \) is \((C/A)\times1\), for any \( E \) and \( w_0 \) and for linearly separable \( T \). To prove the overall result (Theorem[7]), we compose the bound on the length of MCE \( T w_0 \) with the termination refinements from Part I. For further details, see the Coq development. \( \square \)

4.2 Averaged Perceptron

The basic perceptron sometimes produces separators that do not generalize well to unseen data. Variants of the algorithm, like the voted and averaged perceptron, were developed to increase generalizability in exactly such cases [Daume 2017].

For example, instead of returning a single weight vector, the voted perceptron stores all the weight vectors encountered during training, together with how long each “survived” (the number of training vectors between the misclassification that produced the vector and the next misclassification), and makes predictions by tallying each weight vector’s classification votes. The averaged perceptron is a variant of voted perceptron that instead stores just the weighted average of the weight vectors. Both the voted and averaged perceptron use the same convergence criterion as the basic perceptron.

To demonstrate the extensibility of our perceptron implementation and proof, we implemented and proved convergent averaged perceptron on top of our existing codebase. Since the averaged and basic perceptron share the same termination criterion, extending our convergence proof simply required proving an additional termination refinement, in the form of the following theorem:

Lemma perceptron_averaged_perceptron :
\( \forall \langle n : \mathbb{N} \rangle (E : \mathbb{N}) T (w_0 : \mathbb{Q}\text{vec}(S\ n)), \langle w, \text{perceptron} E T w_0 = \text{Some} \ w \rangle \leftrightarrow \langle w, \text{averaged_perceptron} E T w_0 = \text{Some} \ w \rangle. \)

5. Certifier

The perceptron of Listing[2] is a standalone program that both computes a separating hyperplane for \( T \) and checks (in its final iteration of inner_perceptron) that the hyperplane correctly separates \( T \).

In some cases, it may be desirable to run an unverified implementation of perceptron, or even of some other algorithm for learning linear separators, and then merely check that the unverified algorithm produced a valid separator for \( T \). To get soundness guarantees, the checker, or certifier, should itself be proved correct.

We implemented such a certifier by applying just the inner loop of the perceptron algorithm to a purported separator supplied by an oracle, e.g. in C++. In this hybrid architecture, we learn less about the termination behavior of the system (if the oracle is buggy, the program may diverge even if the training data are linearly separable) but still get strong guarantees on soundness (if the certifier succeeds, the purported separator is valid for \( T \)). In situations in which performance is critical but termination bugs are acceptable, this hybrid architecture can give speedups over our fully verified perceptron (Section[7]).

To implement the certifier, we use the inner loop of function perceptron_MCE of Section[2.1.1] (inner_perceptron_MCE) rather than the inner_perceptron of Listing[2]. Both functions return None when the input weight vector correctly classifies \( T \). Upon failure, however, the function inner_perceptron_MCE additionally returns the list of elements that were misclassified, which serves as a counterexample to the purported separator.

We demonstrate soundness of the certifier with the following theorem:

Theorem inner_perceptron_MCE_correctly_classified :
\( \forall \langle n : \mathbb{N} \rangle (E : \mathbb{N}) T (w : \mathbb{Q}vec\ n + \text{bool}), \langle w, \text{inner_perceptron_MCE} T w = \text{None} \rightarrow \text{correctly_classifiedP} T w \rangle. \)

If inner_perceptron_MCE returns None when applying the purported separator \( w \) to training set \( T \), then \( w \) is a valid separator for \( T \) (correctly_classifiedP T w).

6. Fuel for the Fire

In Section[4] we proved (Theorem[7]) that there exists an \( E \) for which perceptron converges, assuming the training set \( T \) is separated by some \( w^* \). But actually producing this \( E \), in order to supply it as fuel to our perceptron program, requires that we first decide whether such a \( w^* \) exists.

It is possible to define and prove a decision procedure for linear separability, e.g., by calculating the convex hulls of the positive and negative labeled instances in \( T \) and checking for intersection. However, we have not yet done so in this work.

Nonetheless, for practical purposes, it is important when running perceptron that we supply fuel large enough not to artificially cause early termination. If the dataset is large, the requisite fuel might be even larger.

In our extracted code, to avoid having to specify very large fuel for large datasets, we instead generate “free fuel” by extracting perceptron to “fueled_perceptron” as follows:

Definition fueled_perceptron :
\( \langle n : \mathbb{N} \rangle (T : \mathbb{N}) (E : \mathbb{N}) T (w : \mathbb{Q}\text{vec}\ n + \text{bool}), \langle \text{fueled_perceptron} E T w = \text{Some} \ w \rangle \leftrightarrow \langle \text{inner_perceptron_MCE} T w = \text{Some} \ w \rangle. \)

The function fuel supplies \( f \) with fuel \( 0 \) in Coq but is extracted to a Haskell function that applies \( f \) to infiniteGas, as generated by the coinductive equation let infiniteGas = S infiniteGas[7] in infiniteGas[7].

6.1 Extraction

In the experiments we will describe in Section[7] we find that judicious use of extraction directives, especially:

- Haskell arbitrary-precision Rationals for Coq \( \mathbb{Q} \)

- Haskell lists for Coq \( \mathbb{Q} \)-vectors

greatly speeds up the Haskell code we extract from our Coq perceptron. Because extraction directives increase the size of our trusted computing base, we briefly justify, in this section, our particular choices.

We extract Coq rationals \( \mathbb{Q} \) to Haskell arbitrary-precision Rationals using the following directive:

Extract Inductive \( \mathbb{Q} \Rightarrow "\text{Rational}" \ ["(n d \rightarrow \text{Data.Ratio.\%} n d)"].

along with others for the various \( \mathbb{Q} \) operators, e.g.:

Extract Constant Qplus \Rightarrow "(Prelude.+)**."

The work of [Pichardie and Bertot 2001] or of [Brun et al. 2012] may be helpful here.

[7] As suggested by an anonymous reviewer, one alternative to the use of “free fuel” is to use Coq’s Acc together with singleton elimination – a strategy which avoids the use of custom extraction directives.
Such directives do not introduce bugs as long as Haskell’s Rationals and associated operators over Rationals correctly implement the Coq Q operators we use in our Coq perceptron. In order to speed up operations over Coq positives and 2s, we use similar Extract Inductive directives to extract both types to Haskell arbitrary-precision Integers.

Our final optimization extracts Coq vectors (Vector.t, or Qvec for the specialization of Vector.t to Q) to Haskell lists, using directives such as:

```
Extract Inductive Vector.t ⇒
  ''([(a_ v → a : v)''
  ''(\V nil fCons w →
    case v of
      [] → fNil ()
      (a : v) → fCons a O v'').''
Extract Constant Coq.Vectors.Vector.fold_left ⇒
  ''(\g a l → Prelude.foldl g a l)''.
```

Taken together, these directives provide big speedups over an unoptimized extraction of the same Coq program (Section 7.1). One obvious speedup comes from using Haskell’s optimized arbitrary-precision Rationals, implemented as pairs of arbitrary-precision Integers. Another is likely from using standard Haskell lists and list functions, such as foldl, which a Haskell compiler such as GHC may optimize more fully than the vector type extracted from Coq.

With extraction directives turned on, our toplevel Haskell perceptron is:

```
type Qvec = ([]) Rational
...
perceptron ::
  Nat → Nat → ([(]) ((,) Qvec Prelude.Bool)) → Qvec →
  Option Qvec
perceptron n e t w =
  case e of
    O → None;
    S e' →
      case inner_perceptron n t w of
        Some w' → perceptron n e' t w';
        None → Some w;
      fueld_perceptron n t w =
        gas (\fuel → perceptron n fuel t w)
```

The function perceptron checks for sufficient fuel e but always reduces to the S case because of its calling context (fueld_perceptron and gas). The type of training data T is no longer a list of Qvecs but rather a list of Rational lists, with each vector paired to a classification label of type Bool.

### 7. Experiments

In this section, we answer a number of experimental questions, including: (1) Is our Coq perceptron practical for real-world use? (The short answer is a qualified “yes” — so far, we’ve had success on some small- to medium-size data sets but have not yet experimented with large data sets.) (2) How well does it scale relative to a C++ implementation of the same algorithm, using arbitrary-precision rationals instead of Coq Q, in number of features, number of training vectors, and size of feature coefficients? (3) How well does it scale relative to a certifier-style implementation with a fast C++ floating-point oracle?

We performed two main experiments. In the first (Section 7.1), we ran our various perceptron implementations on two real-world data sets downloaded from the UCI Machine Learning Repository [Lichman2013]: the classic Iris pattern-recognition data set [Fisher1936] and a “Mines vs. Rocks” data set. Both data sets are known to be linearly separable. In the second experiment (Section 7.2), we generated a number of random linearly separable data sets that differed in number of features, number of training vectors, and the magnitude of the feature coefficients. We consider each experiment in turn.

#### 7.1 Real-World Data Sets

The Iris pattern-recognition data set [Fisher1936] was collected in the 1930s, primarily by Edgar Anderson [Anderson1935] in Québec’s Gaspé peninsula. This data set measures 4 features of 3 species of Iris and includes 150 training vectors. The features are: sepal width, sepal length, petal width, petal length. To turn the 3-class Iris data set into a binary classification problem, we labeled the feature vectors either Iris setosa or not Iris setosa (Iris versicolor or Iris virginica).

Our second real-world data set, also drawn from the UCI Machine Learning Repository, used sonar to discriminate metal cylinders (mines) from rocks across 60 features. Each feature was reported as a number with fixed precision between 0.0 and 1.0. While this data set contained 208 training vectors, it took a considerable number of iterations to converge to a separator.

We report the total runtime (in seconds) of our extracted Coq perceptrons, our arbitrary-precision rational and floating-point C++ implementations, and our extracted Coq validation of the C++ output in Figure 3. We measure two extraction schemes. The first (labeled Coq⇒Haskell) extracts to Haskell using only the extraction directive associated to gas in Section 6; the second (Coq⇒OptHaskell) additionally uses the other extraction directives described in Section 6. Coq Q to the arbitrary-precision Haskell Rational type and Coq Vector.t to Haskell list. Although the “Rocks vs. Mines” data set is small (60 features across 208 instances), it required 275227 epochs to converge; the Iris data set required 4. While C++ outperforms Coq⇒Haskell on both data sets, the Coq⇒OptHaskell outperforms the C++ rational implementation by a factor of 3 on the “Rocks vs. Mines” dataset. The C++ floating-point implementation outperformed all other implementations on both data sets. But while C++ floating-point returned a correct separator for the Iris data set, the separator it returned in Rocks vs. Mines was not always present. The correct separator was not always found.

#### 7.2 Does Coq perceptron Scale?

To experimentally evaluate the asymptotic performance of our Coq perceptrons, we generated a number of linearly separable data sets that differed across number of feature vectors, number of features, and bounds on the magnitude of feature coefficients (to evaluate the overhead of Coq Q). To generate each such data set, we first randomly initialized a separating weight vector w (giving a random separating hyperplane), and then drew feature vectors from a discrete uniform distribution. We labeled each random feature vector as either positive or negative by calculating which side of the separating hyperplane it fell on. To ensure that no feature vectors fell exactly on the separating hyperplane, we rejected those vectors whose dot product with w equaled 0.

Figure 4 displays the results. In the plot, we show the relative runtime of the “vanilla” unoptimized Coq, optimized Coq, validator, and C++ floating-point and arbitrary precision rational perceptrons, normalized to C++ floating-point plus validation. As the number of vectors increases, classification becomes more difficult (many more feature vectors will, in general, lie close to the decision boundary). We see that for optimized Coq, C++ over rational, and vanilla Coq, runtime increases worse than linearly with the number of feature vectors (as expected: the number of epochs required also grew worse than linearly as the number of vectors increased). Our optimized Coq perceptron is about one order of magnitude slower.
than the C++ floating-point implementation, but is on par with (and sometimes faster than) the C++ rational implementation. The vanilla Coq implementation that does not use the extraction directives of Section 6 is a little less than one order of magnitude slower yet again than the optimized Coq implementation.

Perhaps unsurprisingly, although the C++ floating-point implementation with validation in Coq was fastest among all the implementations we tested, it also often generated incorrect separators. On problems of size 50 vectors, for example, it typically misclassified about 60% of vectors.

Our optimized Coq perceptron performed about as well as the C++ rational implementation of the same algorithm, but in some cases surpassed C++ (by about 3x on the real-world Rocks vs. Mines data set of Figure 3, for example). Both the C++ rational and optimized Coq implementations were about 10x slower than a floating-point implementation of perceptron. However, the floating-point perceptron only rarely produced perfect separators on the instances we tested. In some machine-learning contexts, perhaps approximate separators are sufficient. The unoptimized Coq perceptron was dog slow compared to all other implementations. We attribute the slowdown to the use of user-defined Qs and user-defined collection types like Coq Vector.t, which a Haskell compiler such as GHC may not optimize as fully as functions over standard data structures like Haskell lists.

8. Related Work

Bhat [Bhat 2013] has formalized nonconstructive implementations of classic machine-learning algorithms such as expectation maximization in a typed DSL embedded in Coq. However, we are not aware of previous work on mechanized proofs of convergence of such procedures, or of learning procedures in general. A subset of the theorem proving community has embraced machine-learning methods in the design and use of theorem provers themselves (cf. the work on ML4PG [Komendantskaya et al. 2012] for Coq or ACL2(ml) [Heras et al. 2013]).

There is more work on termination for general programs. Theorem provers based on dependent type theory such as Coq and Agda [Norell 2007] generally require that recursive functions be total. Coq uses syntactic guardedness checks whereas provers like Agda incorporate more compositional type-based techniques such as Abel’s sized types [Abel 2004]. There are other ways to prove termination such as Coq’s Program Fixpoint and Function features, or through direct use of Coq Fix. These latter features typically require that the user prove a well-founded order over one of the recursive function’s arguments. The termination argument might be more sophisticated and rely on, e.g., well-quasi-orders [Vytiniotis et al. 2012]. In the automated theorem proving literature, researchers have had success proving termination of nontrivial programs automatically (e.g., [Cook et al. 2006]).

Extending our Q-valued Coq perceptron to use floating-point numbers, following work on floating-point verification in Coq such as Flocco [Boldo and Melquiond 2011] and [Ramananandro et al. 2016], is an interesting next step. Nevertheless, many of the additional research challenges are orthogonal to our results so far. For one, analyzing the behavior of learning algorithms in limited-precision environments is still an active topic in machine learning (cf. [Gupta et al. 2015] for some recent results and a short survey). Nor do we know of any paper-and-pencil perceptron convergence proof that allows for approximation errors due to floating-point computation.

Grégoire, Bertot, and others [Bertot 2015] [Grégoire and Théry 2006] have applied theorem provers such as Coq to computationally intensive numerical algorithms, e.g., computing proved-correct approximations of π to a million digits. We have done initial experiments with Grégoire and Théry’s BigZ/BigQ library (used in both [Bertot 2015] and [Grégoire and Théry 2006]), in the hope that it might speed up our Coq perceptron of Section 7. In initial tests, we’ve seen a slight speedup when switching to BigQ (about 1.6×) with vm_compute in Coq but a slowdown in the extracted OCaml over 1000 iterations of the inner loop of perceptron on vectors of size 2000. The reason is, perhaps, that the representation of Z in Grégoire and Théry’s BigZ was optimized for very large integers and for operations like square root, which is not used by the perceptron inner loop. In fact, we noticed that we could drive the relative speedup of BigQ higher (under vm_compute) by increasing the size of coefficients in the test vectors.

9. Conclusion

This paper presents the first implementation in a theorem prover of the perceptron algorithm and the first mechanically verified proof of perceptron convergence. More broadly, our proof is a case-study application of interactive theorem proving technology to an understudied domain: proving termination of learning procedures. We hope our work spurs researchers to consider other problems in the verification of machine-learning methods, such as (for instance) asymptotic convergence of SVM training algorithms. At the same time, there is still work to do to make verified implementations of such algorithms usable at scale. Our perceptron certifier architecture demonstrates one method for scaling such systems, by composing fast unverified implementations with verified validators.
References
Ulf Norell. Towards a Practical Programming Language Based on Dependent Type Theory, 2007.