Certified Convergent Perceptron Learning

Timothy Murphy* Patrick Gray Gordon Stewart
*Princeton University Ohio University

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 – dependent type theory as implemented in the interactive theorem prover Coq – to prove that one-layer perceptrons for binary classification converge when trained on linearly separable datasets (the Perceptron convergence theorem). We perform experiments to evaluate the performance of our Coq Perceptron vs. a 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 C++ implementation.

Our work is both proof engineering and intellectual archaeology: Even classic machine learning algorithms (and to a lesser degree, termination proofs) are understudied in the interactive theorem proving literature. At the same time, recasting Perceptron and its convergence proof in the language of 21st century human-assisted theorem provers may illuminate, for a fresh audience, a small but interesting corner of the history of ideas.

Keywords machine learning, binary classification, Perceptron, convergence proofs

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

1. Introduction

Frank Rosenblatt – who did work in psychology and neurobiology in addition to computer science – 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 simple boolean predicates. 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 himself[1]) was foundering not because *Perceptrons* drove researchers away, but because the early connectionist work lacked firm theoretical foundation.

[1] Cf. page ix of the 1988 edition of Minsky and Papert's book.

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). We view our work as 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 as reformulation of an existing body of knowledge – with the express goal of making the class of learning procedures the Perceptron represents (stochastic gradient descent) a bit more palatable to researchers in the CPP community.

To this end, we make the following specific contributions:

- We implement Perceptron in Coq.
- We prove that our Coq implementation of Perceptron converges assuming there exists a hyperplane that correctly classifies the training set. Our proof is 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 calculates 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 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 fast 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 at:

`github.com/tm507211/CoqPerceptron`.

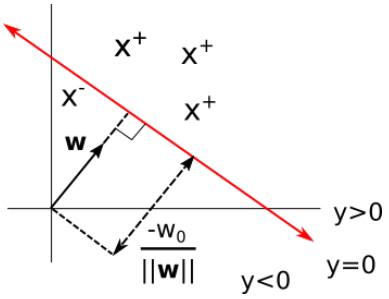


Figure 1. Decision boundary geometry

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 $\mathbf{x} \in \mathbb{R}^n$, where each feature is a numerical delineation of an attribute possibly correlated with the given class label $l \in \{-1, +1\}$. A learned weight model $\mathbf{w} \in \mathbb{R}^n$ 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 $\mathbf{x}$ onto $\mathbf{w}$ and clamps the result as either positive or negative:

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

Figure 1, which we model after (Bishop 2006, Fig. 4.1), represents the decision boundary in two dimensions. Geometrically speaking, the boundary (solid diagonal line from top left to bottom right) is a hyperplane orthogonal to $\mathbf{w}$ with displacement from the origin determined by w_0. All positively classified $\mathbf{x}$ lie on one side of the hyperplane, while all negatively classified $\mathbf{x}$ lie on the other. This formulation of Perceptron easily generalizes to arbitrary dimensions.

The Perceptron learns the decision boundary by minimizing the following error function, known as the

for E epochs or until convergence **do**
  for $(\mathbf{x}, l) \in T$ **do**
    $y = sign(\mathbf{w}^{\mathbb{T}}\mathbf{x})$
    if $y \neq l$ **then**
      $\mathbf{w} = \mathbf{w} + \mathbf{x}l$

Figure 2. Perceptron algorithm

Perceptron criterion:

$$E(\mathbf{w}) = -\sum_{\mathbf{x} \in M} (\mathbf{w}^{\mathbb{T}}\mathbf{x} + w_0)l \tag{2}$$

M is the set of all $\mathbf{x}$ misclassified by $\mathbf{w}$. Since $\mathbf{x}$ is misclassified, the projection of $\mathbf{x}$ onto $\mathbf{w}$ will have sign opposite $\mathbf{x}$'s true label l. Multiplying by the correct label l therefore results in a negative value, forcing the overall error $E(\mathbf{w})$ positive.

To actually minimize $E(\mathbf{w})$, Perceptron performs stochastic gradient descent:

$$\mathbf{w}_k = \mathbf{w}_{(k-1)} - \nabla_{\mathbf{w}} E(\mathbf{w}) = \mathbf{w}_{(k-1)} + \mathbf{x}_k l_k \tag{3}$$

The weight vector at step k is defined to equal $\mathbf{w}_{(k-1)}$ plus the k^{th} misclassified feature vector $\mathbf{x}_k$ multiplied by its class label l_k. The overall effect of this update is to draw the decision boundary closer to the misclassified vector $\mathbf{x}_k$, with the hope that $\mathbf{w}_k$ is now nearer a decision boundary for all the $\mathbf{x}_i$.

The Perceptron algorithm (summarized in pseudocode in Figure 2) will not converge if its training set is inseparable: at least one feature vector will always be misclassified. In the pseudocode of Figure 2, T is the set of labeled training data that guides the Perceptron toward a decision boundary. One iteration of stochastic gradient descent over T is rarely sufficient to find a perfect separator. Thus the Perceptron may execute many times before converging. When T is inseparable, or just not known to be separable, one can force termination by specifying the maximum number of epochs E.

A data set is linearly separable if there exists a weight vector $\mathbf{w}^*$ and bias term w_0^* such that all feature vectors $\mathbf{x}$ in the training data T have predicted sign equal to their true class label l.

Definition 1 (Linear Separability)**.**
Linearly Separable $T \triangleq$
$\exists \mathbf{w}^*.\ \exists w_0^*.\ \forall (\mathbf{x}, l) \in T.\ l = sign(\mathbf{w}^{*\mathbb{T}}\mathbf{x} + w_0^*)$.

3. Perceptron Converges, Informally

As far as we are aware, (Papert 1961) and then (Block 1962) were the first to prove that the Perceptron procedure converges.[2] In this section, we sketch the key ideas underlying the informal proof upon which our formal proof is based (Section 4). The modern informal proof is essentially textbook machine learning – see, for example, (Jaakkola Fall 2006).

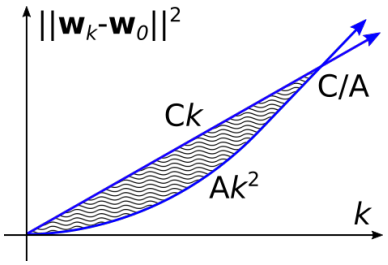


Figure 3. AC Bound

Figure 3 gives an 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 $||\mathbf{w}_k - \mathbf{w}_0||^2$ equal the square of the Euclidean norm of the weight vector (minus the initial weight vector $\mathbf{w}_0$) at that point.[3] The convergence proof proceeds by first proving that $||\mathbf{w}_k - \mathbf{w}_0||^2$ 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 $\mathbf{w}_0$, and the assumed separator $\mathbf{w}^*$.) 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 2 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 $\mathbf{w}_k$ – the weight vector at step k – can be rewritten in terms of $\mathbf{w}_{k-1}$ and the most recently misclassified element $\mathbf{x}_k$ as:

$$\mathbf{w}_k = \mathbf{w}_{k-1} + \mathbf{x}_k \tag{4}$$
$$= \mathbf{w}_0 + \mathbf{x}_1 + \cdots + \mathbf{x}_k \tag{5}$$

Subtracting the initial weight vector $\mathbf{w}_0$ and multiplying both sides by $\mathbf{w}^{*\mathbb{T}}$, the transpose of the assumed separating vector $\mathbf{w}^*$, results in

$$\mathbf{w}^{*\mathbb{T}}(\mathbf{w}_k - \mathbf{w}_0) = \mathbf{w}^{*\mathbb{T}}(\mathbf{x}_1 + \cdots + \mathbf{x}_k) \tag{6}$$

[2] The end of Minsky and Papert's book (Minsky and Papert 1969) includes a much more thorough bibliographic survey of the early literature.

[3] Subtracting $\mathbf{w}_0$ simplifies the calculation of A and C.

Let $a = \min_{\mathbf{x} \in T} \mathbf{w}^{*\mathbb{T}} \mathbf{x}$ be the minimum vector product $\mathbf{w}^{*\mathbb{T}} \mathbf{x}$ across all vectors $\mathbf{x}$ in the training set T. Then

$$\mathbf{w}^{*\mathbb{T}}(\mathbf{w}_k - \mathbf{w}_0) \geq \mathbf{w}^{*\mathbb{T}}(\mathbf{x}_1 + \cdots + \mathbf{x}_k) \quad \geq ak \tag{7}$$

$$||\mathbf{w}^*||^2 ||\mathbf{w}_k - \mathbf{w}_0|| \geq |\mathbf{w}^{*\mathbb{T}}(\mathbf{w}_k - \mathbf{w}_0)|^2 \quad \geq (ak)^2 \tag{8}$$

where 8 follows from 7 and the Cauchy-Schwarz inequality. From 8, it's straightforward to derive that $A = \frac{a^2}{||\mathbf{w}^*||^2}$.

C is derived by a similar set of inequalities:

$$||\mathbf{w}_k - \mathbf{w}_0||^2 = ||\mathbf{w}_{k-1} + \mathbf{x}_k - \mathbf{w}_0||^2$$
$$= ||\mathbf{w}_{k-1} - \mathbf{w}_0||^2 + 2(\mathbf{w}_{k-1} - \mathbf{w}_0)^{\mathbb{T}} \mathbf{x}_k + ||\mathbf{x}_k||^2 \tag{9}$$

where 9 follows by foiling the square of the Euclidean norm. Assuming $\forall i \in [1, k], \mathbf{w}_{i-1}^{\mathbb{T}} \mathbf{x}_i \leq 0$, which holds after each $\mathbf{x}_i$ has been normalized by multiplying it by its class label, we get that

$$||\mathbf{w}_k - \mathbf{w}_0||^2$$
$$\leq ||\mathbf{w}_{k-1} - \mathbf{w}_0||^2 - 2(\mathbf{w}_0^{\mathbb{T}} \mathbf{x}_k) + ||\mathbf{x}_k||^2 \tag{10}$$
$$\leq ||\mathbf{x}_1||^2 + \cdots + ||\mathbf{x}_k||^2 - 2\mathbf{w}_0^{\mathbb{T}}(\mathbf{x}_2 + \cdots + \mathbf{x}_k) \tag{11}$$

Inequality 10 follows from 9 and from nonpositivity. Summing over all i = 1 to k gives 11 from 10.

Define $M = \max_{\mathbf{x} \in T} ||\mathbf{x}||^2$ and $\mu = \min_{\mathbf{x} \in T} \mathbf{w}_0^{\mathbb{T}} \mathbf{x}$. Then from inequality 11 we have that

$$||\mathbf{w}_k - \mathbf{w}_0||^2 \leq Mk - 2\mu k = (M - 2\mu)k \tag{12}$$

giving $C = M - 2\mu$.

4. Implementation and Formal Proof

Now we turn to the Coq implementation and formal convergence 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 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 $\mathbf{w}$. Thus our $\mathbf{w}$ vectors are of size $\#features + 1$.

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

Listing 1. Basic Definitions

```
Definition ℚvec ≜ Vector.t ℚ.
Definition class (i : ℚ) : bool ≜ ℚle_bool 0 i.
Definition correct_class (i : ℚ) (l : bool) : bool ≜
  Bool.eqb l (class i) && negb (ℚeq_bool i 0).
Definition consb {n : nat} (v : ℚvec n) ≜ 1 :: v.
```

ℚvec is just an abbreviation for the Coq standard-library vector type specialized to $\mathbb{Q}$. The class (or sign) of an input i : $\mathbb{Q}$ (as produced, for example, from an input vector $\mathbf{x}$ by $\mathbf{w}^{\mathbb{T}}\mathbf{x} + w_0$) is determined by checking whether i is greater than or equal to 0. We say the input i is correctly classified, according to label l, if l equals class i and i is nonzero. This second condition forces our Coq Perceptron to continue working until no feature vectors lie on the decision boundary.

Listing 2 gives the main definitions.

Listing 2. Coq Perceptron

```
Fixpoint inner_percepton {n : nat}
  (T : list (ℚvec n*bool) (w : ℚvec n.+1)
  : option (ℚvec n.+1) ≜
  match T with nil ⇒ None
   | (x, l) :: T' ⇒
     if correct_class (ℚvec_dot w (consb x)) l
     then inner_perceptron T' w
     else
      let wx ≜
       ℚvec_plus w (ℚvec_mult_class l (consb x))
      in match inner_perceptron T' wx with
           | None ⇒ Some wx
           | Some w' ⇒ Some w'
         end end.

Fixpoint perceptron {n : nat} {E : nat}
  (T : list (ℚvec n*bool)) (w : ℚvec n.+1)
  : option (ℚvec n.+1) ≜
  match E with 0 ⇒ None
   | S E' ⇒
     match inner_perceptron T w with
      | None ⇒ Some w
      | Some w' ⇒ perceptron E' T w'
     end end.
```

The fixpoint inner_perceptron, which corresponds to the inner loop in Figure 2, 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 $\mathbf{w}$.[4] Upon correct classification of an $\mathbf{x}$, inner_perceptron simply moves to the next training vector. Upon misclassification, we let the new decision vector, $\mathbf{w}$x, equal the vector sum of $\mathbf{w}$ and $\mathbf{x}$ multiplied by its class label. The function then continues iterating through the remaining training samples T′.

Listing 2's second fixpoint, perceptron, implements the outer loop of Figure 2. Its recursion parameter is run-of-the-mill fuel E, a natural number 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 $\mathbf{w}'$. By the definition of perceptron, convergence only occurs when the algorithm has settled on a $\mathbf{w}$ that correctly classified all the training vectors in T (inner_perceptron T $\mathbf{w}$ = None). Thus soundness (if perceptron converges, it does so with a vector $\mathbf{w}'$ that separates the training set T) is trivial.

Listing 3. Linear Separability

```
Definition correctly_classifiedP {n : nat}
  : list (ℚvec n∗bool) → ℚvec n.+1 → Prop ≜
  λT w ⇒ List.Forall
   (λ xl : (ℚvec n ∗ bool) ⇒
     let (x, l) ≜ xl
     in 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*.
```

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 $\mathbf{w}$

states that vector $\mathbf{w}$ correctly classified training set T (a list of vector-label pairs). A data set T is linearly separable when there exists a $\mathbf{w}^*$ such that $\mathbf{w}^*$ correctly classifies T. The main convergence result is thus:

Theorem 1 (Perceptron Converges)**.**
∀{*n:nat*} *(T : list (ℚvec n∗bool)) (*$\mathbf{w}_0$ *: ℚvec n.+1),*

[4] The function consb (Listing 1) conses 1 to $\mathbf{x}$ to account for $\mathbf{w}$'s bias term.

```
Fixpoint MCE {n:nat} (E:nat)
 (T : list (ℚvec n∗bool)) (w : ℚvec n.+1)
 : list (ℚvec n∗bool) ≜ ...

                ⊑⇓

Fixpoint perceptron_MCE {n:nat} (E:nat)
 (T : list (ℚvec n∗bool)) (w : ℚvec n.+1)
 : option (list (ℚvec n∗bool) ∗ ℚvec n.+1) ≜ ...

                ⊑⇓

Fixpoint perceptron {n : nat} {E : nat}
 (T : list (ℚvec n∗bool)) (w : ℚvec n.+1)
 : option (ℚvec n.+1) ≜ ...
```

Figure 4. Alternative Perceptrons. The symbol $\sqsubseteq_{\Downarrow}$ denotes termination refinement.

```
linearly_separable T ↔
∃(w : ℚvec n.+1) (E₀ : nat),
∀E : nat, E ≥ E₀ →
  perceptron E T w₀ = Some w.
```

For all training sets T and initial vectors $\mathbf{w}_0$, T is linearly separable iff there is an E_0 such that perceptron converges (to some separator $\mathbf{w}$) when run on E_0 or greater fuel. This theorem trivially subsumes the case when $\mathsf{E} = \mathsf{E}_0$.

4.1 Formal Convergence Proof

What about the formal proof? By the definition of perceptron, the ($\leftarrow$) direction of Theorem 1 is easy. But as we outlined in Section 3, the ($\rightarrow$) direction (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 $\mathbf{w}'$ (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 a handle on these "misclassified elements" in our formal proof, we defined two alternative **Fixpoint**s as depicted in Figure 4. 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 :
  ∀(n E : nat)
   (w₀ : ℚvec n.+1) (T : list (ℚvec n * bool)),
  MCE E T w₀ = MCE E.+1 T w₀ →
  perceptron_MCE E.+1 T w₀
    = Some (MCE E T w₀,
            ℚvec_sum_class w₀ (MCE E T w₀)).

Lemma perceptron_MCE_eq_perceptron :
  ∀(n E : nat)
   (T : list (ℚvec n * bool)) (w₀ w : ℚvec n.+1),
  (∃ M : list (ℚvec n * bool),
     perceptron_MCE E T w₀ = Some (M, w)) ↔
  perceptron E T w₀ = 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 $\mathbf{w}_0$ has reached a fixed point (the list of misclassified feature vectors is stable regardless 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 $\mathbf{w}_0$ and each vector in MCE E T $\mathbf{w}_0$ multiplied by its class label.

The second lemma proves an equitermination property (subsuming the refinement shown in Figure 4): the function perceptron_MCE converges to Some (M, $\mathbf{w}$) on training set T iff perceptron also converges to $\mathbf{w}$.

4.1.2 Part II.

The main difficulty in part II is proving the AC bound on the length of MCE E T $\mathbf{w}_0$, the number of misclassified feature vectors. Note that the length of MCE E T $\mathbf{w}_0$ need not be less than or equal to $|\mathsf{T}|$, 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 1 (MCE Bounded)**.**

```
∀{n:nat} (T : list (ℚvec n * bool)) (w₀ : ℚvec n.+1),
 linearly_separable T →
 ∃A B C : nat, A ≠ 0 ∧ B ≠ 0 ∧ C ≠ 0 ∧
 ∀E : nat, A * |MCE E T w₀|²
   ≤ B * ℚvec_normsq (ℚvec_sum (MCE E T w₀))
   ≤ C * |MCE E T w₀|.
```

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 : ∀{n:nat} (x₁ x₂ : ℚvec n),
 ℚvec_dot x₁ x₂ * ℚvec_dot x₁ x₂ ≤
 ℚvec_normsq x₁ * ℚvec_normsq x₂.
```

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

We use Lemma 1 to prove the overall convergence result (Theorem 1).

Proof of Theorem 1 (Perceptron Convergence). Lemma 1 and the following arithmetic fact

$$\forall xyz : \mathbb{N}.\ x \neq 0 \wedge y \neq 0 \wedge z > y/x \ \Rightarrow\ xz^2 > yz$$

together imply that the maximum length of MCE E T $\mathbf{w}_0$ is (C/A).+1, for any E and $\mathbf{w}_0$ and for linearly separable T. To prove the overall result (Theorem 1), we compose the bound on the length of MCE E T $\mathbf{w}_0$ with the termination refinements from **Part I**. For further details, see the Coq development. □

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, as Section 7 demonstrates.

To implement the certifier, we use the inner loop of perceptron_MCE of Figure 4 (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 :
  ∀n (T : list (ℚvec n*bool)) (w : ℚvec n. + 1),
    inner_perceptron_MCE T w = None →
    correctly_classifiedP T w.
```

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 1) 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.[5] However, we have not yet done so in this work.

[5] The work of (Pichardie and Bertot 2001) or of (Brun et al. 2012) may be helpful if we choose to formalize this algorithm in the future.

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, especially when represented as nat rather than binary ℤ.

In our extracted code, to avoid having to specify very large fuel for large datasets, we instead generate "free fuel" (a trick suggested by an anonymous reviewer) by extracting perceptron to "fueled_perceptron" as follows:

```
Definition gas (T : Type) (f : nat → T) : T ≜ f O.

Extract Constant gas ⇒
 "(\f → let infiniteGas = S infiniteGas
        in f infiniteGas)".

Definition fueled_perceptron
  (n _ : nat)
  (T : list (ℚvec n * bool)) (w : ℚvec (S n))
  : option (ℚvec (S n)) ≜
    gas (λ fuel ⇒ perceptron fuel T w).
```

The function gas supplies f with fuel 0 but is extracted to the Haskell function that applies f to infiniteGas, as generated by the equation let infiniteGas = S infiniteGas.

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 ℚs
- Haskell lists for Coq 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 ℚ to Haskell arbitrary-precision Rationals using the following directive:

```
Extract Inductive ℚ ⇒ "Rational"
  [ "(\n d → (Data.Ratio.%) n d)" ].
```

along with others for the various ℚ operators, e.g.:

```
Extract Constant ℚplus ⇒ "(Prelude.+)".
```

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 $\mathbb{Z}$s, we use similar **Extract Inductive** directives to extract both types to Haskell arbitrary-precision Integers.

Our final optimization extracts Coq vectors (Vector.t, or $\mathbb{Q}$vec for the specialization of Vector.t to $\mathbb{Q}$) to Haskell lists, using directives such as:

```
Extract Inductive Vector.t ⇒
  ''([])'' [ ''[]'' ''(\a _ v → a : v)'' ]
  ''(\fNil fCons v →
      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, as Section 7 will demonstrate. 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 and functions 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}}

fueled_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 (fueled_perceptron and gas). The type of training data T is no longer a list of $\mathbb{Q}$vecs but rather a list of [Rational], with each vector paired to a classification label of type Bool.

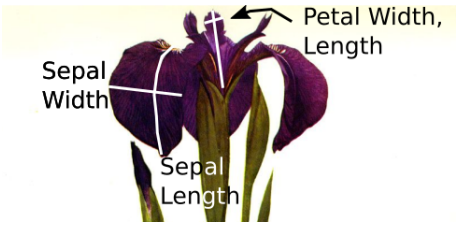


Figure 5. Iris setosa (watercolor credit F. H. Round, Plate XXIII from (Dykes et al. 1913))

7. Experiments

It is the authors' hope that theorem provers such as Coq (or Isabelle/HOL (Nipkow et al. 2002), or HOL4 (Slind and Norrish 2008), or others – we are ecumenical) will one day be the IDEs of choice for more than just the most discerning programmers. But to build that reality, we must first be honest about the limits of programming and proving in functional languages with clean proof theories. For example, what is the performance overhead of a data-centric computation such as our Coq perceptron (when extracted to Haskell and compiled using ghc) over a re-implementation of the same algorithm in C++, using STL arbitrary-precision rational numbers? What is the overhead against the certifier-style architecture of Section 5, in which the separator oracle is implemented by a fast C++ implementation using floating-point numbers? We would expect that C++ handily beats Coq in this domain, but by how much?

In this section, we answer these questions and others, including:

- Is our Coq perceptron practical for use on real-world data? (The short answer is "yes" – so far, we've had success on some small- to medium-size data sets.)
- How well does it scale relative to a C++ implementation of the same algorithm, using arbitrary-precision rationals instead of Coq $\mathbb{Q}$, in number of features, number of training vectors, and size of feature coefficients?
- How well does it scale relative to a certifier-style implementation with a fast C++ floating-point oracle?

To answer these questions, we performed two 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 Reposi-

	Coq⇒Haskell	**Coq⇒$_{\mathsf{Opt}}$Haskell**	**C++ Rational**	**C++ FP**	**Validator**
Iris	$0.049s$	$0.027s$	$0.039s$	$0.021s$	$0.027s$
Rocks vs. Mines	$95.4h$	$2.14h$	$6.56h$	$48.787s$	$0.295s$

Figure 6. Coq vs. Coq-Optimized vs. arbitrary-precision rational and floating-point C++ on real-world data

tory (Lichman 2013): the classic Iris pattern-recognition data set (Fisher 1936) 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 (Fisher 1936) was collected in the 1930s, primarily by Edgar Anderson (Anderson 1935) in Québec's Gaspé peninsula. This data set measures 4 features of 3 species of Iris (Figure 5) 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* (the species depicted in Figure 5) 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 on a separator.

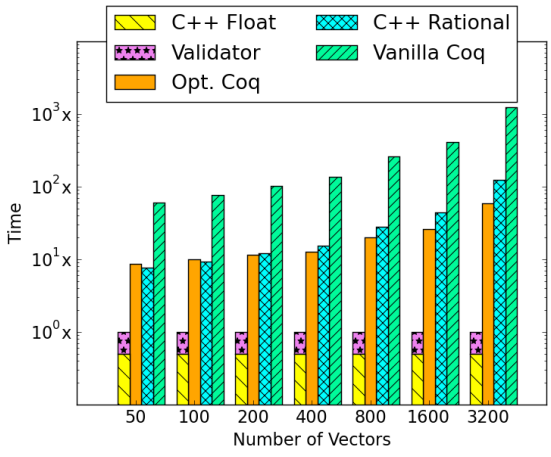
We report the total runtime (in seconds) of our extracted Coq Perceptrons, our arbitrary-precision rational and floating-point C++ implementations, and our extracted Coq validation of the C++ output in Figure 6. We measure two extraction schemes. The first (labeled **Coq⇒Haskell**) extracts to Haskell using only the extraction directive associated to `gas` in Section 6; the second (**Coq⇒$_{\mathsf{Opt}}$Haskell**) additionally uses the other extraction directives described in Section 6 – Coq $\mathbb{Q}$ to the arbitrary-precision Haskell `Rational` type and Coq `Vector.t` to Haskell `list`. Although the "Rocks vs. Mines" data set is of only moderate size (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⇒$_{\mathsf{Opt}}$Haskell** 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 misclassified 2 of 208 training vectors, as a result of floating-point approximation errors.

7.2 Does Coq perceptron Scale?

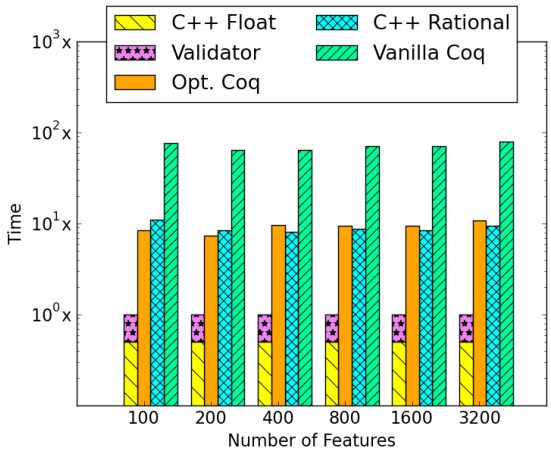
To experimentally evaluate the asymptotic performance of our Coq Perceptrons, we generated a number of linearly separable data sets that differed across number of feature vectors, number of features, and bounds on the magnitude of feature coefficients (to evaluate the overhead of Coq $\mathbb{Q}$). To generate each such data set, we first randomly initialized a separating weight vector $\mathbf{w}$ (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 $\mathbf{w}$ equaled 0.

Figure 7 displays the results. In each subplot, we show the relative runtime of the "vanilla" unoptimized Coq, optimized Coq, validator, and C++ floating-point and arbitrary precision rational Perceptrons on each of three synthetic data sets that vary in number of vectors included in the classification problem (Figure 7a), in number of features per vector (Figure 7b), and in magnitude of feature coefficients (Figure 7c). We normalize each experiment to a baseline ($10^0 x$): C++ floating point followed by validation by the extracted Coq validator. The plots are in log scale.

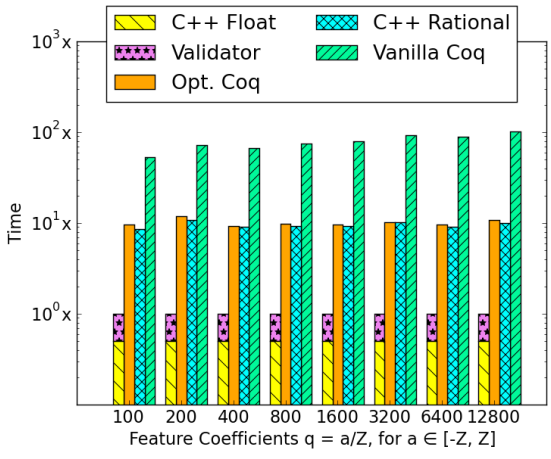
Figure 7a displays the results of the vector experiment. As the number of vectors increases, classification becomes more difficult (many more feature vectors will, in general, lie close to the decision boundary). We see that for optimized Coq, C++ over rationals, and vanilla Coq, runtime increases worse than linearly with the number of feature vectors (as expected: the number of



(a)



(b)



(c)

Figure 7. Coq vs. optimized Coq vs. rational and floating-point C++ on three synthetic data sets

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.

Although the C++ floating-point implementation with validation in Coq was fastest among all the implementations we tested, it often generated incorrect separators. Figure 8 summarizes the misclassification results for the vectors experiment of Figure 7a. The **%Errors** row of the table in Figure 8 lists the percentage of randomly generated problem instances of each class (50 vectors, 100 vectors, etc.) in which the C++ floating-point implementation generated separators that misclassified at least one feature vector in the training set. As the number of vectors grows, and thus the difficulty of the classification problems increases, the number of misclassifications increases as well. We noticed similar percentages of misclassification errors by the C++ floating-point implementation in the other two synthetic experiments.

To validate that our C++ floating-point Perceptron was generating separating vectors that were at least approximately correct, we also calculated square cosine similarity for each vector with respect to the vector produced by our Coq implementation (**%Similar.** in Figure 8). We see, perhaps counterintuitively, that as classification difficulty increases, similarity also increases. The reason is, more difficult problem instances require many more iterations of the Perceptron loop, which drives the approximate separator produced C++ floating-point closer and closer to the true separator.

Figures 7b and 7c show results of the two additional experiments. In the first (Figure 7b), we ran each Perceptron on randomly generated data sets with increasing numbers of features. In general, the presence of more features makes the linear discrimination problem easier; we controlled for this by fixing the number of feature vectors to just 100, which ensured that the number of epochs was approximately constant across all runs. The vanilla Coq Perceptron is almost an order of magnitude slower than the optimized Coq and C++ rational Perceptrons, and is nearly two orders of magnitude slower than the C++ floating-point implementation.

#Vectors	50	100	200	400	800	1600	3200
%Errors	60	60	70	100	90	100	100
%Similarity	96.89	98.34	99.14	99.54	99.78	99.90	99.97

Figure 8. Percentage of problem instances (**%Errors**) in which the C++ floating-point implementation generated imperfect separators in the vectors experiment of Figure 7a; square cosine similarity of the C++ floating-point separators (**%Similarity**) with respect to the vectors produced by our Coq implementation.

In the second additional experiment (Figure 7c), we generated data sets with ever larger expected feature values. The x-axis of Figure 7c gives bounds. For example, the first set of data points gives runtimes on a data set with 1000 features, 100 feature vectors, and feature values $q = a/Z$, with Z drawn uniformly from the interval $[-100, 100]$. Although some rational features may not be in reduced form when generated in this way, we still expect the average size of features to grow as Z increases. In the plot, the C++ rational and optimized Coq runtimes are approximately constant relative to the C++ floating-point implementation, whereas the vanilla Coq implementation, which uses extracted Coq $\mathbb{Q}$, grows slower relative to C++ floating-point as the size of feature coefficients increases.

Overall, we were quite surprised by how well our optimized Coq Perceptron performed relative to the C++ rational implementation of the same algorithm, in some cases surpassing C++ (by about $3x$ on the real-world Rocks vs. Mines data set of Figure 6, 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 $\mathbb{Q}$s 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 Haskell lists.

8. Related Work

We are not aware of previous work on mechanized proof of convergence of learning procedures. Bhat (Bhat 2013), however, has formalized nonconstructive implementations of classic machine-learning algorithms such as expectation maximization in a typed DSL embedded in Coq. 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. As is well known, theorem provers based on dependent type theory such as Coq and Agda (Norell 2007) require for consistency that all recursive functions be total. Coq uses syntactic guardedness checks whereas provers like Agda now incorporate more compositional type-based techniques such as Abel's sized types (Abel 2004). There are other ways to prove termination, of course, 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 (and do not work well when the function, such as our Coq `perceptron`, is total on only a subet of inputs). 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 $\mathbb{Q}$-valued Coq perceptron to use floating-point numbers, following work on floating-point verification in Coq such as Flocq (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 vanilla Coq Perceptron of Section 7. In initial tests, we've seen a slight speedup when switching to BigQ (about $1.6\times$) 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 $\mathbb{Z}$ 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, as far as we are aware, the first mechanically verified proof of the Perceptron Convergence Theorem. More broadly, our proof is a case-study application of ITP technology to an understudied domain: proving termination of learning procedures. We hope our work spurs researchers to consider other challenging 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 the Coq implementations of such algorithms usable at scale.

References

A. Abel. Termination Checking with Types. *ITA*, 2004.

E. Anderson. The Irises of the Gaspé Peninsula. *Bulletin of the American Iris Society*, 59:2–5, 1935.

Y. Bertot. Fixed precision patterns for the formal verification of mathematical constant approximations. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 147–155. ACM, 2015.

S. Bhat. *Syntactic foundations for machine learning*. PhD thesis, Georgia Institute of Technology, 2013.

C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

H.-D. Block. The Perceptron: A Model for Brain Functioning. I. *Reviews of Modern Physics*, 34(1):123, 1962.

S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pages 243–252. IEEE, 2011.

C. Brun, J.-F. Dufourd, and N. Magaud. Formal Proof in Coq and Derivation of an Imperative Program to Compute Convex Hulls. In *Aut. Deduction in Geom.* 2012.

B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code. In *PLDI*, 2006.

W. R. Dykes et al. *The Genus Iris*. Cambridge, 1913.

R. A. Fisher. The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics*, 7(2):179–188, 1936.

B. Grégoire and L. Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In *International Joint Conference on Automated Reasoning*, pages 423–437. Springer, 2006.

S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. *CoRR, abs/1502.02551*, 392, 2015.

J. Heras, E. Komendantskaya, M. Johansson, and E. Maclean. Proof-pattern Recognition and Lemma Discovery in ACL2. In *LPAR*, 2013.

T. Jaakkola. Course Materials for 6.867 Machine Learning. MIT OpenCourseWare (`http://ocw.mit.edu/`), Massachusetts Institute of Technology, Fall 2006.

E. Komendantskaya, J. Heras, and G. Grov. Machine Learning in Proof General: Interfacing Interfaces. *arXiv preprint arXiv:1212.3618*, 2012.

M. Lichman. UCI Machine Learning Repository, 2013. URL `http://archive.ics.uci.edu/ml`.

M. Minsky and S. Papert. *Perceptrons*. MIT Press, 1969.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

U. Norell. Towards a Practical Programming Language Based on Dependent Type Theory, 2007.

S. Papert. Some mathematical models of learning. In *Proceedings of the Fourth London Symposium on Information Theory*, 1961.

D. Pichardie and Y. Bertot. Formalizing convex hull algorithms. In *International Conference on Theorem Proving in Higher Order Logics*, pages 346–361. Springer, 2001.

T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin. A unified coq framework for verifying c programs with floating-point computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 15–26. ACM, 2016.

F. Rosenblatt. The Perceptron – A Perceiving and Recognizing Automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, 1957.

F. Rosenblatt. *Principles of Neurodynamics; Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1962.

K. Slind and M. Norrish. A Brief Overview of HOL4. In *TPHOLs*. 2008.

The Coq Development Team. The Coq Proof Assistant. `https://coq.inria.fr/`, 2016. [Online; accessed 2-19-2016].

D. Vytiniotis, T. Coquand, and D. Wahlstedt. Stop When You Are Almost-Full. In *ITP*. 2012.