The perceptron algorithm was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt,[3] funded by the United States Office of Naval Research.[4] The perceptron was intended to be a machine, rather than a program, and while its first implementation was in software for the IBM 704, it was subsequently implemented in custom-built hardware as the "Mark 1 perceptron". This machine was designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors.[2]:193

In a 1958 press conference organized by the US Navy, Rosenblatt made statements about the perceptron that caused a heated controversy among the fledgling AI community; based on Rosenblatt's statements, The New York Times reported the perceptron to be "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."[4]
Recap: perceptrons
The perceptron

Compute: $y_i = \text{sign}(v_k \cdot x_i)$

If mistake: $v_{k+1} = v_k + y_i x_i$

Margin $\gamma$. $A$ must provide examples that can be separated with some vector $u$ with margin $\gamma > 0$, ie

$$\exists u: \forall (x_i, y_i) \text{ given by } A, (u \cdot x)y_i > \gamma$$

and furthermore, $\|u\| = 1$.

Radius $R$. $A$ must provide examples “near the origin”, ie

$$\forall x_i \text{ given by } A, \|x\|^2 < R^2$$
The perceptron

**Instance** \( x_i \)

**Compute:** \( y_i = \text{sign}(v_k \cdot x_i) \)

**If mistake:** 
\[
V_{k+1} = V_k + y_i x_i
\]

A lot like SGD update for logistic regression!

\[
W_{(t+1)} = W_{(t)} + \lambda (y - \hat{y}) x
\]

**Mistake bound:**

\[
k \leq \frac{R^2}{\gamma^2} = \left( \frac{R}{\gamma} \right)^2
\]
On line to batch learning

1. Pick a \( v_k \) at random according to \( \frac{m_k}{m} \), the fraction of examples it was used for.
2. Predict using the \( v_k \) you just picked.
3. (Actually, use some sort of deterministic approximation to this).

Imagine we run the on-line perceptron and see this result.

<table>
<thead>
<tr>
<th>( i )</th>
<th>guess</th>
<th>input</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( v_0 )</td>
<td>( x_1 )</td>
<td>X (a mistake)</td>
</tr>
<tr>
<td>2</td>
<td>( v_1 )</td>
<td>( x_2 )</td>
<td>( \checkmark ) (correct!)</td>
</tr>
<tr>
<td>3</td>
<td>( v_1 )</td>
<td>( x_3 )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>4</td>
<td>( v_1 )</td>
<td>( x_4 )</td>
<td>X (a mistake)</td>
</tr>
<tr>
<td>5</td>
<td>( v_2 )</td>
<td>( x_5 )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>6</td>
<td>( v_2 )</td>
<td>( x_6 )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>7</td>
<td>( v_2 )</td>
<td>( x_7 )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>8</td>
<td>( v_2 )</td>
<td>( x_8 )</td>
<td>X</td>
</tr>
<tr>
<td>9</td>
<td>( v_3 )</td>
<td>( x_9 )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>10</td>
<td>( v_3 )</td>
<td>( x_{10} )</td>
<td>X</td>
</tr>
</tbody>
</table>

\[ P(\text{error in } x) = \sum_k P(\text{error on } x|\text{picked } v_k)P(\text{picked } v_k) \]
\[ = \sum_k \frac{1}{m_k} \frac{m_k}{m} = \sum_k \frac{1}{m} = \frac{k}{m} \]
1. Pick a $\mathbf{v}_k$ at random according to $m_k/m$, the fraction of examples it was used for.

2. Predict using the $\mathbf{v}_k$ you just picked.

3. (Actually, use some sort of deterministic approximation to this).

Imagine we run the on-line perceptron and see this result.

<table>
<thead>
<tr>
<th>$i$</th>
<th>guess</th>
<th>input</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\mathbf{v}_0$</td>
<td>$x_1$</td>
<td>$X$ (a mistake)</td>
</tr>
<tr>
<td>2</td>
<td>$\mathbf{v}_1$</td>
<td>$x_2$</td>
<td>$\checkmark$ (correct!)</td>
</tr>
<tr>
<td>3</td>
<td>$\mathbf{v}_1$</td>
<td>$x_3$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>4</td>
<td>$\mathbf{v}_1$</td>
<td>$x_4$</td>
<td>$X$ (a mistake)</td>
</tr>
<tr>
<td>5</td>
<td>$\mathbf{v}_2$</td>
<td>$x_5$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>6</td>
<td>$\mathbf{v}_2$</td>
<td>$x_6$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>7</td>
<td>$\mathbf{v}_2$</td>
<td>$x_7$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>8</td>
<td>$\mathbf{v}_2$</td>
<td>$x_8$</td>
<td>$X$</td>
</tr>
<tr>
<td>9</td>
<td>$\mathbf{v}_3$</td>
<td>$x_9$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>10</td>
<td>$\mathbf{v}_3$</td>
<td>$x_{10}$</td>
<td>$X$</td>
</tr>
</tbody>
</table>
predict using $\text{sign}(v^*. x)$

$$v_* = \sum_k \left( \frac{m_k}{m} v_k \right)$$

Also: there’s a sparsification trick that makes learning the averaged perceptron fast

![Graph showing test error over epochs for different methods: random (unnorm), last (unnorm), avg (unnorm), vote. The graph indicates that averaging/voting performs better than the last perceptron.]
KERNELS AND PERCEPTRONS
The kernel perceptron

Compute: $y_i = \mathbf{v}_k \cdot \mathbf{x}_i$

Compute: $\hat{y} = \sum_{x_{k_+} \in FN} \mathbf{x}_i \cdot \mathbf{x}_{k_+} - \sum_{x_{k_-} \in FP} \mathbf{x}_i \cdot \mathbf{x}_{k_-}$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$

If false positive (too high) mistake: add $\mathbf{x}_i$ to FP
If false positive (too low) mistake: add $\mathbf{x}_i$ to FN

Mathematically the same as before ... but allows use of the *kernel trick*
The kernel perceptron

\[ K(x, x_k) \equiv x \cdot x_k \]

Compute: \[ \hat{y}_i = \sum_{x_{k+} \in \text{FN}} K(x_i, x_{k+}) - \sum_{x_{k-} \in \text{FP}} K(x_i, x_{k-}) \]

If mistake: \[ v_{k+1} = v_k + y_i x_i \]
If false positive (too high) mistake: add \( x_i \) to FP
If false positive (too low) mistake: add \( x_i \) to FN

Mathematically the same as before … but allows use of the “kernel trick”

Other kernel methods (SVM, Gaussian processes) aren’t constrained to limited set (+1/-1/0) of weights on the \( K(x,v) \) values.
Some common kernels

• Linear kernel: $K(x, x') \equiv x \cdot x'$

• Polynomial kernel: $K(x, x') \equiv (x \cdot x' + 1)^d$

• Gaussian kernel: $K(x, x') \equiv e^{-\|x-x'|^2 / \sigma}$
Some common kernels

- Polynomial kernel:

\[ K(x, x') \equiv (x \cdot x' + 1)^d \]

- for \( d=2 \)

\[
\left( \langle x_1, x_2 \rangle \cdot \langle x'_1, x'_2 \rangle + 1 \right)^2 \\
= (x_1 x'_1 + x_2 x'_2 + 1)^2 \\
= (x_1 x'_1 + x_2 x'_2 + 1)(x_1 x'_1 + x_2 x'_2 + 1) \\
= (x_1 x'_1)^2 + 2(x_1 x'_1 x_2 x'_2) + 2(x_1 x'_1) + (x_2 x'_2)^2 + 2(x_2 x'_2) + 1 \\
\cong \langle 1, x_1, x_2, x_1 x_2, x_1^2, x_2^2 \rangle \cdot \langle 1, x'_1, x'_2, x'_1 x'_2, x'_1^2, x'_2^2 \rangle \\
= \langle 1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1 x_2, x_1^2, x_2^2 \rangle \cdot \langle 1, \sqrt{2}x'_1, \sqrt{2}x'_2, \sqrt{2}x'_1 x'_2, x'_1^2, x'_2^2 \rangle \]
Some common kernels

• Polynomial kernel:

\[ K(x, x') \equiv (x \cdot x' + 1)^d \]

• for \( d=2 \)

\[
\left( \langle x_1, x_2 \rangle \cdot \langle x'_1, x'_2 \rangle + 1 \right)^2
\]

\[
= \langle 1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2 \rangle \cdot \langle 1, \sqrt{2}x'_1, \sqrt{2}x'_2, \sqrt{2}x'_1x'_2, x'_1^2, x'_2^2 \rangle
\]

Similarity with the kernel on \( x \) is equivalent to dot-product similarity on a transformed feature vector \( \phi(x) \)
Explicitly map from $x$ to $\phi(x)$ – i.e. to the point corresponding to $x$ in the Hilbert space (RKHS)

Implicitly map from $x$ to $\phi(x)$ by changing the kernel function $K$

- **Duality**: two ways to look at this
  
  **Compute**: $y_i = v_k \cdot x_i$
  
  \[ w = \sum_{x_{k+} \in FN} x_{k+} - \sum_{x_{k-} \in FP} x_{k-} \]

  **Observation about perceptron**
  
  \[ \hat{y} = \phi(x) \cdot w \]
  
  \[ w = \sum_{x_{k+} \in FN} \phi(x_{k+}) - \sum_{x_{k-} \in FP} \phi(x_{k-}) \]

  **Generalization of perceptron**
  
  \[ \hat{y} = \sum_{x_{k+} \in FN} K(x_i, x_{k+}) - \sum_{x_{k-} \in FP} K(x_i, x_{k-}) \]
  
  \[ K(x, x_k) \equiv \phi(x) \cdot \phi(x_k) \]

  **Generalization**: add weights to the sums for $w$

  **same behavior but compute time/space are different**
A FAMILIAR KERNEL: THE HASH KERNEL
Learning as optimization for regularized logistic regression + hashes

• Algorithm:
  \[ w^j = w^j + \lambda(y - p)x^j - \lambda2\mu w^j \]
• Initialize arrays \( W, A \) of size \( R \) and set \( k=0 \)
• For each iteration \( t=1,\ldots,T \)
  – For each example \( (x_i, y_i) \)
    • \( V \) is a hash table
    • For \( j: x^j > 0 \) increment \( V[h[j]] \) by \( x^j \)
    • \( p_i = \ldots ; k++ \)
• For each hash value \( h: V[h] > 0 \):
  » \( W[h] *= (1 - \lambda2\mu)^{k-A[h]} \)
  » \( W[h] = W[h] + \lambda(y_i - p^i)V[h] \)
  » \( A[h] = k \)

\[ V = \varphi(x) \]

\( V \) is a kernel!
Hash Kernels

Qinfeng Shi, James Petterson
Australian National University and NICTA,
Canberra, Australia

John Langford, Alex Smola, Alex Strehl
Yahoo! Research
New York, NY and Santa Clara, CA, USA

Gideon Dror
Department of Computer Science
Academic College of Tel-Aviv-Yaffo, Israel

Vishy Vishwanathan
Department of Statistics
Purdue University, IN, USA
Some details

Slightly different hash to avoid systematic bias

\[ V[h] = \sum_{j: \text{hash}(j) \% R == h} x_i^j \]

\[ \varphi[h] = \sum_{j: \text{hash}(j) \% m == h} \xi(j)x_i^j, \quad \text{where } \xi(j) \in \{-1, +1\} \]

\( m \) is the number of buckets you hash into (R in my discussion)
Technical claims in Shi et al:

• This fancy hash kernel is unbiased:

\[ E_\phi[\langle x, x' \rangle_\phi] = \langle x, x' \rangle. \]

• For L2—normalized \( x, x' \), the variance is:

\[ O(\frac{1}{m}). \]

• Theorem: “with high probability a kernel classifier with this kernel will have similar performance to a (normal) linear kernel. “
The Voted Perceptron for Ranking and Structured Classification
The voted perceptron for ranking

\[ y_i = \mathbf{v}_k \cdot \mathbf{x}_i \]

Return: the index \( b^* \) of the “best” \( \mathbf{x}_i \)

If mistake: \[ \mathbf{v}_{k+1} = \mathbf{v}_k + \mathbf{x}_b - \mathbf{x}_{b^*} \]

Margin \( \gamma \). \( A \) must provide examples that can be correctly ranked with some vector \( \mathbf{u} \) with margin \( \gamma > 0 \), ie

\[ \exists \mathbf{u} : \forall \mathbf{x}_{i,1}, \ldots, \mathbf{x}_{i,n_t}, \ell \text{ given by } A, \forall j \neq \ell, \mathbf{u} \cdot \mathbf{x}_\ell - \mathbf{u} \cdot \mathbf{x}_j > \gamma \]

and furthermore, \( \|\mathbf{u}\|^2 = 1 \).

Radius \( R \). \( A \) must provide examples “near the origin”, ie

\[ \forall \mathbf{x}_i \text{ given by } A, \|\mathbf{x}\|^2 < R^2 \]
Ranking some $x$'s with the target vector $u$
Ranking some x’s with some guess vector v – part 1
Ranking some $x$’s with some guess vector $v$ – part 2.

The purple-circled $x$ is $x_{b^*}$ - the one the learner has chosen to rank highest. The green circled $x$ is $x_b$, the right answer.
Correcting $\mathbf{v}$ by adding $x_b - x_{b^*}$
Correcting $\mathbf{v}$ by adding $x_b - x_b^*$

(part 2)
(3a) The guess $v_2$ after the two positive examples: $v_2 = v_1 + x_2$

Lemma 1 $\forall k$, $v_k \cdot u \geq k\gamma$. In other words, the dot product between $v_k$ and $u$ increases with each mistake, at a rate depending on the margin $\gamma$.

Proof:

$$v_{k+1} \cdot u = (v_k + y_i x_i) \cdot u$$

$$\Rightarrow v_{k+1} \cdot u = (v_k \cdot u) + y_i (x_i \cdot u)$$

$$\Rightarrow v_{k+1} \cdot u \geq v_k \cdot u + \gamma$$

$$\Rightarrow v_k \cdot u \geq k\gamma$$
(3a) The guess \( v_2 \) after the two positive examples: \( v_2 = v_1 + x_2 \)

\[
(3a) \quad \text{The guess } v_2 \text{ after the two positive examples: } v_2 = v_1 + x_2
\]

**Lemma 1** \( \forall k, v_k \cdot u \geq k \gamma \). In other words, the dot product between \( v_k \) and \( u \) increases with each mistake, at a rate depending on the margin \( \gamma \).

\[
\begin{align*}
\begin{array}{l}
v_{k+1} \cdot u & = (v_k + y_i x_i) \cdot u \\
\Rightarrow & v_{k+1} \cdot u = (v_k \cdot u) + y_i (x_i \cdot u) \\
\Rightarrow & v_{k+1} \cdot u \geq v_k \cdot u + \gamma \\
\Rightarrow & v_k \cdot u \geq k \gamma \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
v_{k+1} \cdot u & = (v_k + x_{i,\ell} - x_{i,\ell}) \cdot u \\
\Rightarrow & v_{k+1} \cdot u = v_k \cdot u + x_{i,\ell} \cdot u - x_{i,\ell} \cdot u \\
\Rightarrow & v_{k+1} \cdot u \geq v_k \cdot u + \gamma \\
\Rightarrow & v_k \cdot u \geq k \gamma
\end{array}
\end{align*}
\]
(3a) The guess $v_2$ after the two positive examples: $v_2 = v_1 + x_2$

Lemma 3. $\forall k$, $v_k \cdot u \geq k\gamma$. In other words, the dot product between $v_k$ and $u$ increases with each mistake, at a rate depending on the margin $\gamma$.

\[
\begin{align*}
v_{k+1} \cdot u &= (v_k + y_i x_i) \cdot u \\
\Rightarrow v_{k+1} \cdot u &= (v_k \cdot u) + y_i (x_i \cdot u) \\
\Rightarrow v_{k+1} \cdot u &\geq v_k \cdot u + \gamma \\
\Rightarrow v_k \cdot u &\geq k\gamma
\end{align*}
\]
(3a) The guess $v_2$ after the two positive examples: $v_2 = v_1 + x_2$

Notice this doesn’t depend at all on the number of $x$’s being ranked.

Lemma 4 $\forall k, \|v_k\|^2 \leq 2kR$.

Theorem 2 Under the rules of the ranking perceptron game, it is always the case that $k < 2R/\gamma^2$.

Neither proof depends on the dimension of the $x$’s.
Ranking perceptrons ➔ structured perceptrons

• The API:
  – A sends B a (maybe huge) set of items to rank
  – B finds the single best one according to the current weight vector
  – A tells B which one was actually best

• Structured classification on a sequence
  – Input: list of words: \( x=(w_1,\ldots,w_n) \)
  – Output: list of labels: \( y=(y_1,\ldots,y_n) \)
  – If there are K classes, there are \( K^n \) labels possible for \( x \)
Inference for linear-chain CRFs

When will prof Cohen post the notes ...

Idea 1: features are properties of \textit{two adjacent tokens}, and the \textit{pair} of labels assigned to them (Begin,Inside,Outside)

- $(y(i)==B \text{ or } y(i)==I)$ and (token(i) is capitalized)
- $(y(i)==I \text{ and } y(i-1)==B)$ and (token(i) is hyphenated)
- $(y(i)==B \text{ and } y(i-1)==B)$
  
  • eg “tell Rose William is on the way”

Idea 2: construct a graph where each \textit{path} is a possible sequence labeling.
Inference for a linear-chain CRF

• Inference: find the highest-weight path given a weighting of features
• This can be done efficiently using dynamic programming (Viterbi)
Ranking perceptrons ➔ structured perceptrons

• New API:
  – A sends B the word sequence \( x \)
  – B finds the single best \( y \) according to the current weight vector using Viterbi
  – A tells B which \( y \) was actually best

  – This is equivalent to ranking pairs \( g = (x, y') \)

• Structured classification on a sequence
  – Input: list of words: \( x = (w_1, \ldots, w_n) \)
  – Output: list of labels: \( y = (y_1, \ldots, y_n) \)
  – If there are K classes, there are \( K^n \) labels possible for \( x \)
The voted perceptron *for ranking*

Compute: $y_i = \mathbf{v}_k \cdot \mathbf{x}_i$

Return: the index $b^*$ of the "best" $\mathbf{x}_i$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + \mathbf{x}_b - \mathbf{x}_{b^*}$

Change number one is notation: replace $\mathbf{x}$ with $\mathbf{g}$
The voted perceptron for structured classification tasks

Compute: \( y_i = \hat{v}_k \cdot g_i \)

Return: the index \( b^* \) of the “best” \( g_i \)

If mistake: \( v_{k+1} = v_k + g_b - g_{b^*} \)

1. A sends B feature functions, and instructions for creating the instances \( g \):
   - A sends a word vector \( x_i \). Then B could create the instances \( g_1 = F(x_i, y_1), g_2 = F(x_i, y_2), \ldots \)
   - but instead B just returns the \( y^* \) that gives the best score for the dot product \( v_k \cdot F(x_i, y^*) \) by using Viterbi.

2. A sends B the correct label sequence \( y_i \).

3. On errors, B sets \( v_{k+1} = v_k + g_b - g_{b^*} = v_k + F(x_i, y) - F(x_i, y^*) \)
Results from the original paper....


Michael Collins
AT&T Labs-Research, Florham Park, New Jersey.
mcollins@research.att.com

EMNLP 2002, Best paper
Collins’ Experiments

• POS tagging
• NP Chunking (words and POS tags from Brill’s tagger as features) and BIO output tags
• Compared logistic regression methods (MaxEnt) and “Voted Perceptron trained HMM’s”
  – With and w/o averaging
  – With and w/o feature selection (count>5)
## Collins’ results

### NP Chunking Results

<table>
<thead>
<tr>
<th>Method</th>
<th>F-Measure</th>
<th>Numits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perc, avg, cc=0</td>
<td>93.53</td>
<td>13</td>
</tr>
<tr>
<td>Perc, noavg, cc=0</td>
<td>93.04</td>
<td>35</td>
</tr>
<tr>
<td>Perc, avg, cc=5</td>
<td>93.33</td>
<td>9</td>
</tr>
<tr>
<td>Perc, noavg, cc=5</td>
<td>91.88</td>
<td>39</td>
</tr>
<tr>
<td>ME, cc=0</td>
<td>92.34</td>
<td>900</td>
</tr>
<tr>
<td>ME, cc=5</td>
<td>92.65</td>
<td>200</td>
</tr>
</tbody>
</table>

### POS Tagging Results

<table>
<thead>
<tr>
<th>Method</th>
<th>Error rate/%</th>
<th>Numits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perc, avg, cc=0</td>
<td>2.93</td>
<td>10</td>
</tr>
<tr>
<td>Perc, noavg, cc=0</td>
<td>3.68</td>
<td>20</td>
</tr>
<tr>
<td>Perc, avg, cc=5</td>
<td>3.03</td>
<td>6</td>
</tr>
<tr>
<td>Perc, noavg, cc=5</td>
<td>4.04</td>
<td>17</td>
</tr>
<tr>
<td>ME, cc=0</td>
<td>3.4</td>
<td>100</td>
</tr>
<tr>
<td>ME, cc=5</td>
<td>3.28</td>
<td>200</td>
</tr>
</tbody>
</table>

**Figure 4:** Results for various methods on the part-of-speech tagging and chunking tasks on development data. All scores are error percentages. Numits is the number of training iterations at which the best score is achieved. Perc is the perceptron algorithm, ME is the maximum entropy method. Avg/noavg is the perceptron with or without averaged parameter vectors. cc=5 means only features occurring 5 times or more in training are included, cc=0 means all features in training are included.
Debugging Machine Learning Algorithms

William Cohen
Debugging for non-ML systems

• “If it compiles, ship it.”
Debugging for ML systems

1. It’s definitely *exactly* the algorithm you read about in that paper
2. It also compiles
3. It gets 87% accuracy on the author’s dataset
   - but he got 91%
   - so it’s not working?
   - or, your eval is wrong?
   - or, *his* eval is wrong?
Debugging for ML systems

1. It’s definitely *exactly* the algorithm you read about in that paper
2. It also compiles
3. It gets 97% accuracy on the author’s dataset
   – but he got 91%
   – so you have a best paper award!
   – or, maybe a bug...
Debugging for ML systems

• It’s always hard to debug software
• It’s especially hard for ML
  – a wide range of almost-correct modes for a program to be in
EMPIRICAL RESEARCH
It’s easy to make assumptions about puppies strapped to missiles, but good science requires testing.
Debugging advice

1. Write tests
2. For subtle problems, write tests
3. If you’re still not sure why it’s not working, write tests
4. If you get really stuck:
   - take a walk and come back to it in a hour
   - ask a friend
     •  If s/he’s also in 10-605 s/he can still help as long as no notes are taken (my rules)
   - take a break and write some tests
Debugging ML systems

1. Write tests
   – For a generative learner, write a generator and generate training/test data from the assumed distribution
     • Eg, for NB: use one small multinomial for pos examples, another one for neg examples, and a weighted coin for the class priors.
   – The learner should (usually) recover the actual parameters of the generator
     • given enough data, modulo convexity, ...
   – Test it on the weird cases (eg, uniform class priors, highly skewed multinomials)
The importance of math

• Math helps us design good learners
• It also helps us predict what they should do
• You can use that when you write tests
Debugging ML systems

1. Write tests
   – For a discriminative learner, similar trick...
   – Also, use what you know: eg, for SGD
     • does taking one gradient step (on a sample task) lower the loss on the training data?
     • does it lower the loss as expected?
       – \((f(x)-f(x+d))/d\) should approximate \(f'(x)\)
     • does regularization work as expected?
       – large \(\mu\) \(\Rightarrow\) smaller param values
     • record training set/test set loss
       – with and without regularization