Linear Multi-Class Classification

Instructor: Hui Guan

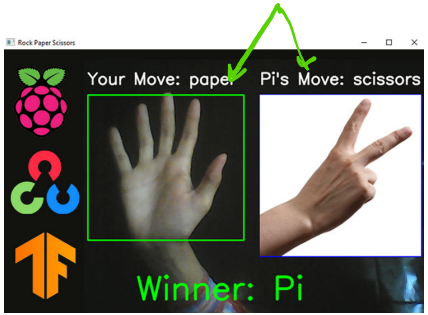
Slides adapted from: https://github.com/jermwatt/machine_learning_refined

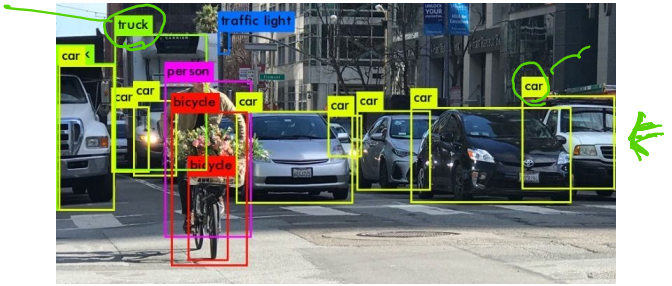
Week	Dates	Class	Topic	Reading	Assignments	Note - University Calendar
	9/5	Lecture 1	Intro	Ch. 1		First Day of classes
1	9/7	Lecture 2	Optimization - Global	Ch. 2		
	9/12	Lecture 3	Optimization - Local		- HW1 on random search out	Monday, Sept.11th: Last day to add or drop any class with no record – Undergraduate and Non-degree students
2	9/14	Lecture 4	Optimization - Gradient Descent	Ch. 3		
	9/19	Lecture 5	Optimization - Gradient Descent (cont.)		- HW2 on gradient decent out.	Monday, Sept. 18th: Last day to add or drop any class with no record – Matriculated Graduate students
3	9/21	Lecture 6	Linear regression	Ch. 5	- HW1 DUE.	
	9/26	Lecture 7	Linear classification - logistic regression	Ch. 6		
4	9/28	Lecture 8	Linear classification - perceptron		- HW2 DUE. - HW3 on linear regression out.	
	10/3	Lecture 9	Linear classification - SVM			
5	today 10/5	Lecture 10	Linear classification - SVM cont. + multiclass	Ch. 7	- HW 3 DUE - HW 4 on logistic regression out	
☆	10/10	NO CLASS				Monday Oct. 9th: Holiday – Indigenous Peoples Day; Tuesday Oct. 10th: Monday class schedule will be followed.
6	10/12	Lecture 11	Unsupervised - PCA	Ch. 8	- HW 4 DUE - HW 5 on multi-class classification out	
	10/17	Lecture 12	Unsupervised - Kmeans			
7	10/19	Lecture 13	Feature engineering	Ch. 9		

Week	Dates	Class	Topic	Reading	Assignments	Note - University Calendar
	10/24	Lecture 14	Feature engineering		- HW 5 DUE - HW6 on unsupervised learning out	
8	10/26	Lecture 15	Nonlinear learning	Ch. 10		
	10/31	Lecture 16	Feature learning - part 1	Ch. 11	- HW 6 DUE - HW 7 on feature engineering out	Tuesday, Oct. 31st: - Last day to Drop with "DR" - Graduate; - Last day to Drop with 'W' and select 'P/F' - Undergraduate, Stockbridge, CPE
9	11/2	Lecture 17	Feature learning - part 2			
	11/7	Lecture 18	Kernel methods - part 1	Ch. 12		
10	11/9	Lecture 19	Midterm Recap		- HW7 DUE - HW8 feature learning out	
	11/14	Lecture 20	Kernel methods - part 2			
11	11/16	Lecture 21	**Midterm (In Person)**			
	11/21	Lecture 22	NO CLASS		- HW8 DUE - Project out	Tuesday, Nov. 21st: Thanksgiving recess begins after last class
12	11/23	NO CLASS				
	11/28	Lecture 23	MLP - part 1	Ch. 13		
13	11/30	Lecture 24	MLP - part 2			
	12/5	Lecture 25	Tree - part 1	Ch. 14		
14	12/7	Lecture 26	Tree - part 2		- Project DUE	Last Class for Machine Learning
	12/8					Friday, Dec. 8th: Last day of classes
	12/11					Monday, Dec. 11: Final examinations begin
	12/15					Friday, Dec. 15: Last day of final exam, semester ends
	12/21					Final grades due by Midnight

Many classification problems have more than two classes:

- face recognition
- hand gesture recognition
- general object detection
- speech recognition
- …





Having just two sides, a single linear separator is fundamentally insufficient as a mechanism for differentiating between more than two classes of data.

Outline

- One-versus-All Multi-Class Classification
- Multi-class classification and the Perceptron
- Which approach produces the best results?
- Classification Quality Metrics
- Mini-batch training

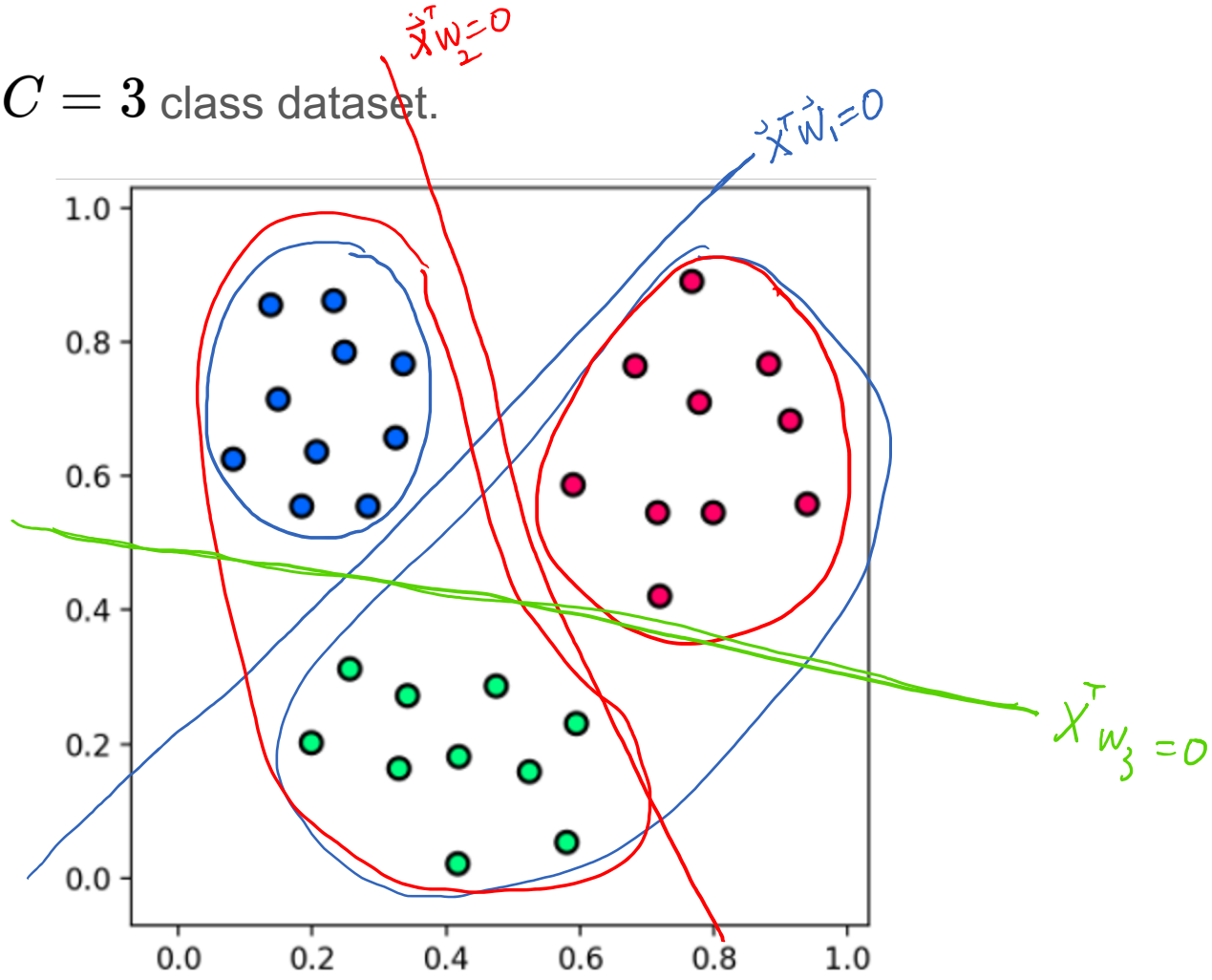
7.2 One-versus-All Multi-Class Classification

A multiclass dataset $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$

consists of C distinct classes of data.

Although we can in theory use *any C distinct labels* to distinguish between the classes, it is convenient to use label values $y_p \in \{0, 1, \ldots, C-1\}$

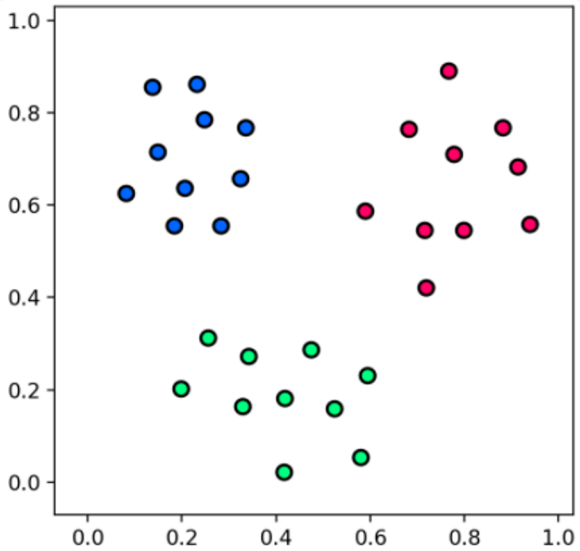
A prototypical toy $C = 3$ class dataset.



Can you use 2-class classification model for this problem?

If so, how would you build your classifiers?

Given a new point, how to make decision on the predicted class label of the point?



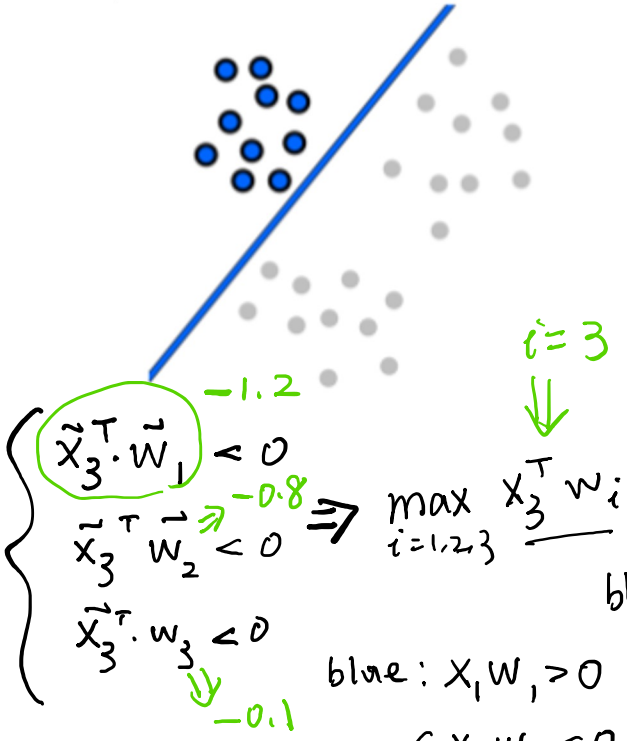
Training ‘C’ One-versus-All classifiers

The goal:

We want to learn how to distinguish each class of our data from the other C-1 classes.

One solution:

Learn C two-class classifiers on the entire dataset (with the c^{th} classifier trained to distinguish the c^{th} class from the remainder of the data).

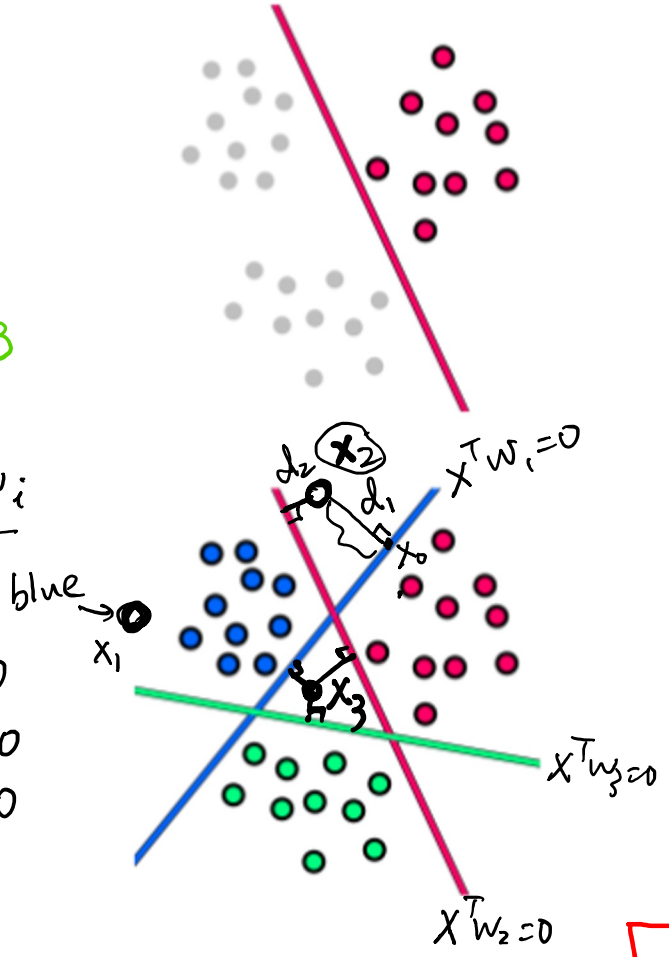


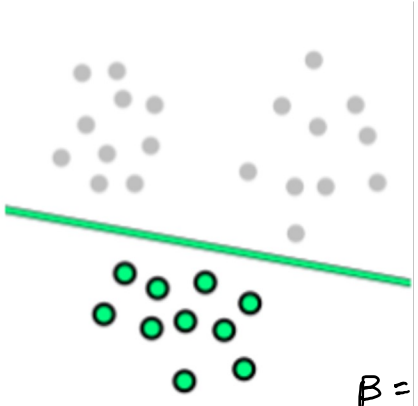
$\Rightarrow \max_{i=1,2,3} x_3^T w_i$

$i = 3$

blue : $x_1 w_1 > 0$

$\begin{cases} x_1 w_2 < 0 \\ x_1 w_3 < 0 \end{cases}$





$d_1 = \|x_2 - x_0\|_2 = \dfrac{\beta = x_2^T w_1}{\|w_1\|_2}$

$\left.\begin{matrix} x_2^T w_1 = \beta \\ x_0^T w_1 = 0 \end{matrix}\right\} \Rightarrow (x_2 - x_0)^T w_1 = \beta$

$\|x_2 - x_0\|_2 \cdot \|w_1\|_2 = \beta$

$d_2 = \dfrac{\gamma = x_2^T w_2}{\|w_2\|_2}$

$d_1(x_2) > d_2(x_2)$

$\max x_2^T w_i \quad , i \in \{1, 2, 3\}$

The three places a point in the space can end up (w.r.t our classifiers):

1. **On the positive side of a single classifier**
2. **On the positive side of more than one classifier**
3. **On the negative side of all classifiers**

Points on the positive side of a single classifier

Denoting the weights from the c^{th} classifier as $\mathbf{w}_c$

$$\mathbf{w}_c = \begin{bmatrix} w_{0,c} \\ w_{1,c} \\ w_{2,c} \\ \vdots \\ w_{N,c} \end{bmatrix}$$

then the corresponding decision boundary can be written as

$$\dot{\mathbf{x}}^T \mathbf{w}_c = 0.$$

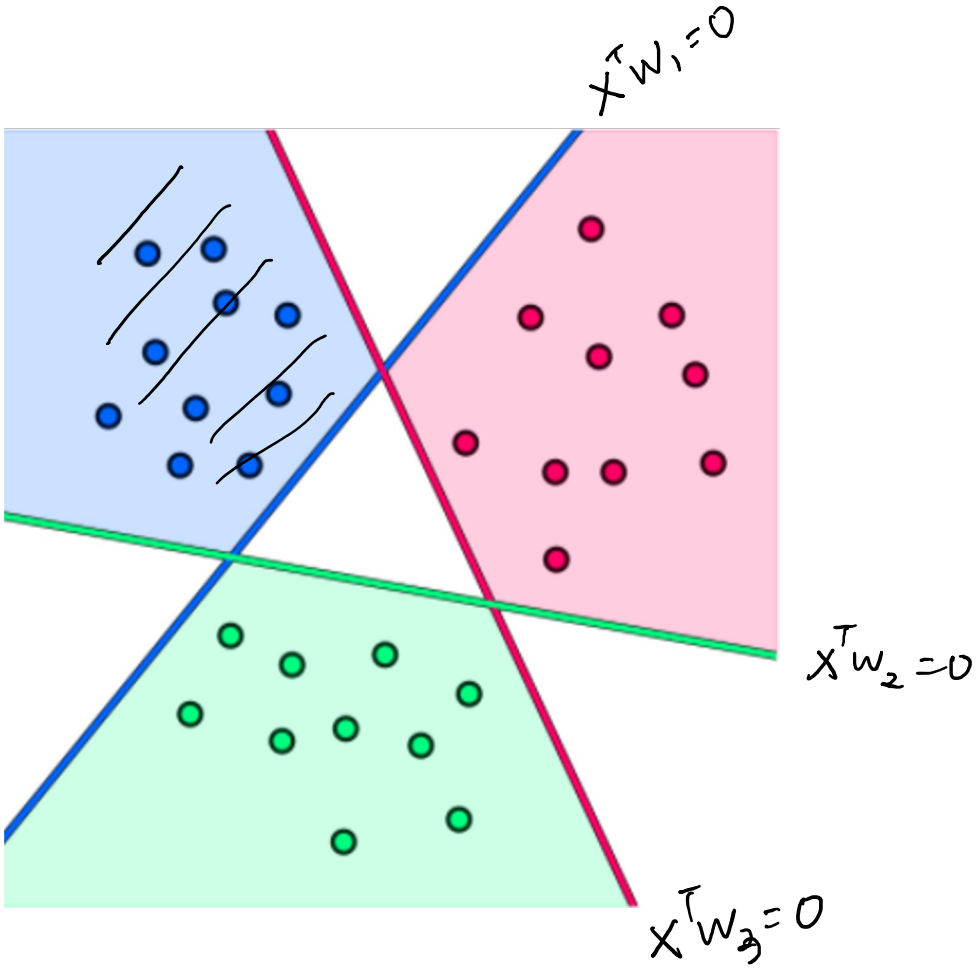
The points that lie solely on the positive side of the j^{th} classifier only should clearly belong to the j^{th} class.

They clearly satisfy the condition

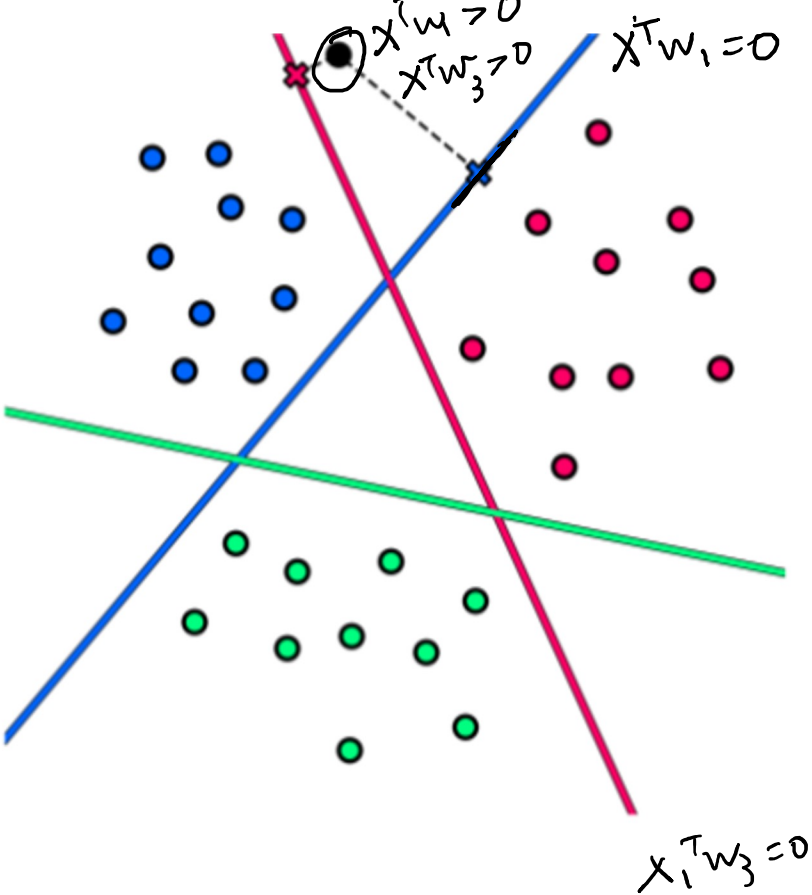
$$\dot{\mathbf{x}}^T \mathbf{w}_j = \max_{c=0,\ldots,C-1} \dot{\mathbf{x}}^T \mathbf{w}_c$$

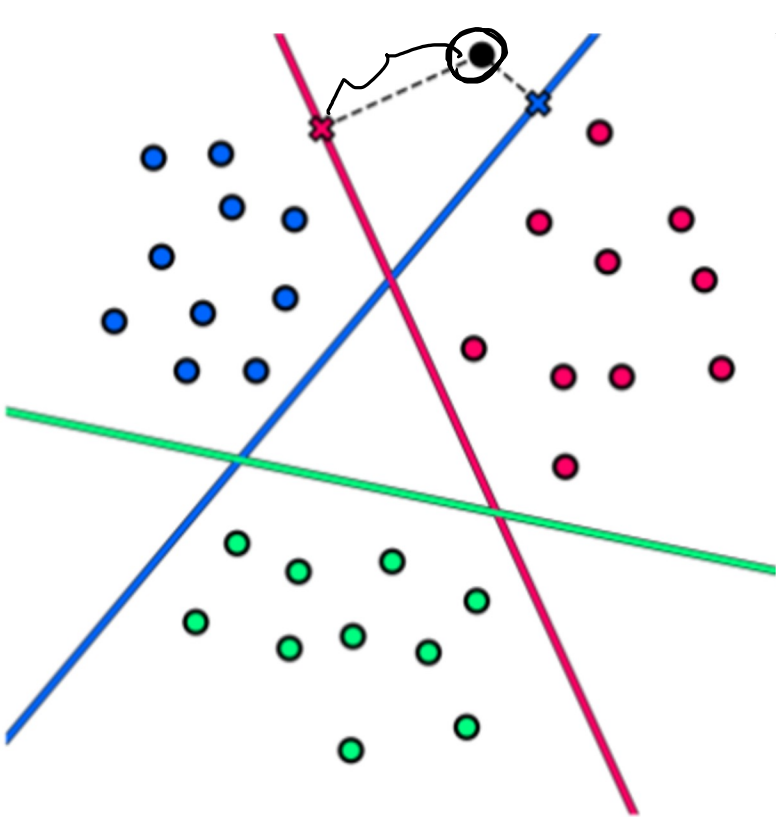
Therefore to get the associated label y for these points, we can write

$$y = \operatorname*{argmax}_{c=0,\ldots,C-1} \dot{\mathbf{x}}^T \mathbf{w}_c$$


$x^T w_1 = 0$
$x^T w_2 = 0$
$x^T w_3 = 0$

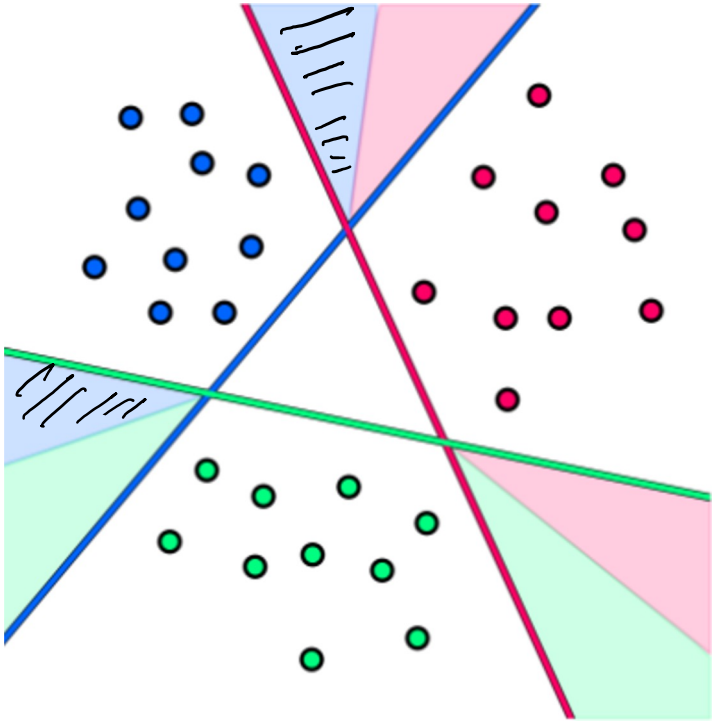
Points on the positive side of more than one classifier





Which class should we assign each point to?

If we repeat this logic for every point on the positive side of more than two classifiers ...



Classification rule: we assign each point to the class whose boundary is at the largest nonnegative distance from it.

How do we compute the distance of an arbitrary point to each of our decision boundaries?

$$\text{signed distance of } \mathbf{x} \text{ to } j^{th} \text{ boundary} = \frac{\dot{\mathbf{x}}^T \mathbf{w}_j}{\|\boldsymbol{\omega}_j\|_2} \sim \dot{\mathbf{x}}^T \mathbf{w}_j$$

"normalized"

$$\mathbf{w}_j = \frac{\mathbf{w}_j}{\|\mathbf{w}_j\|_2}$$

If we *normalize* the weights of each linear classifier by the length of its normal vector as

$$\mathbf{w}_j \longleftarrow \frac{\mathbf{w}_j}{\left\|\boldsymbol{\omega}_j\right\|_2}$$

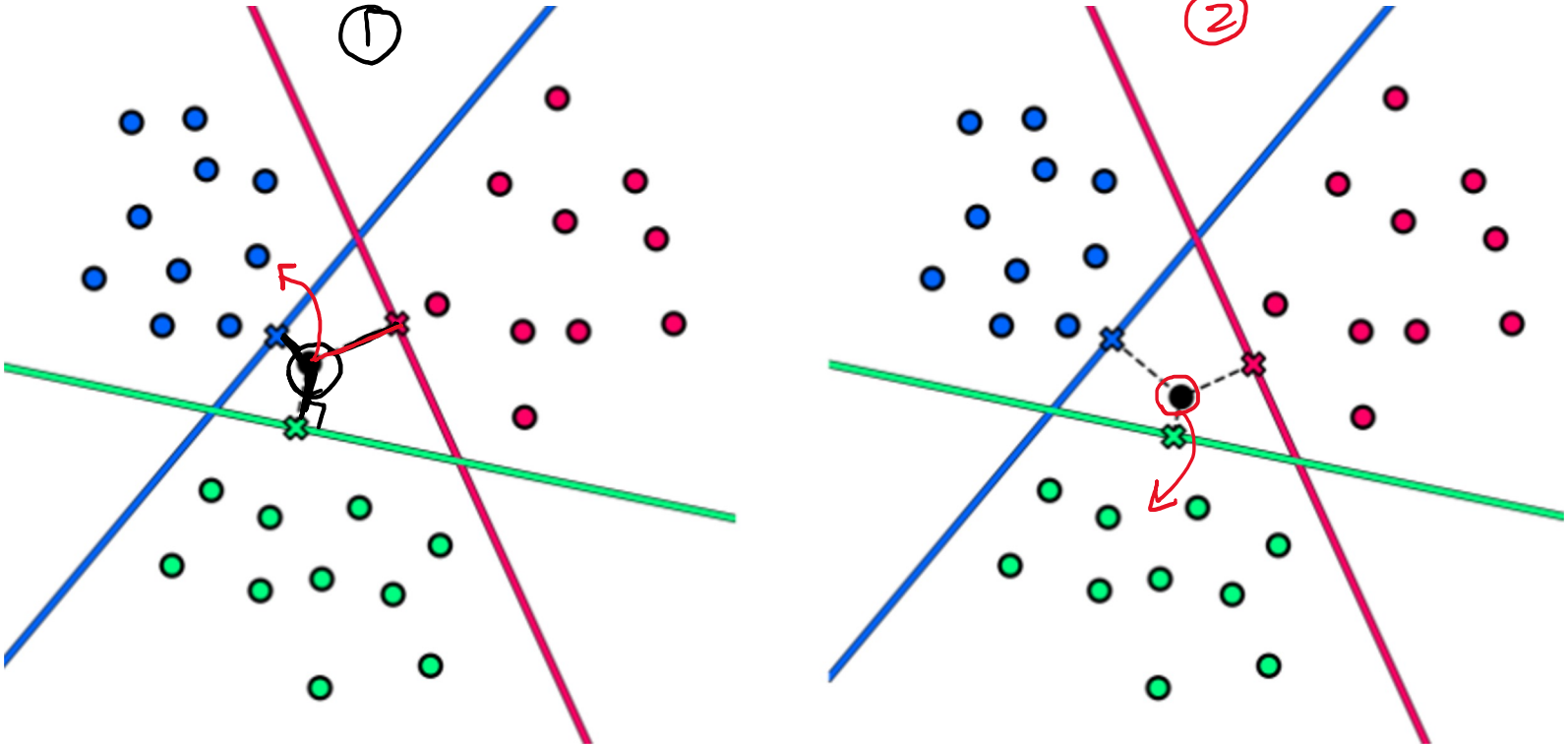
then this distance is simply written as the raw evaluation of the point via the decision boundary as

$$\text{signed distance of } \mathbf{x} \text{ to } j^{th} \text{ boundary} = \dot{\mathbf{x}}^T \mathbf{w}_j$$

Therefore again - after weight-normalization - we have precisely the same prediction rule we found originally for regions of the space where only a single classifier is positive

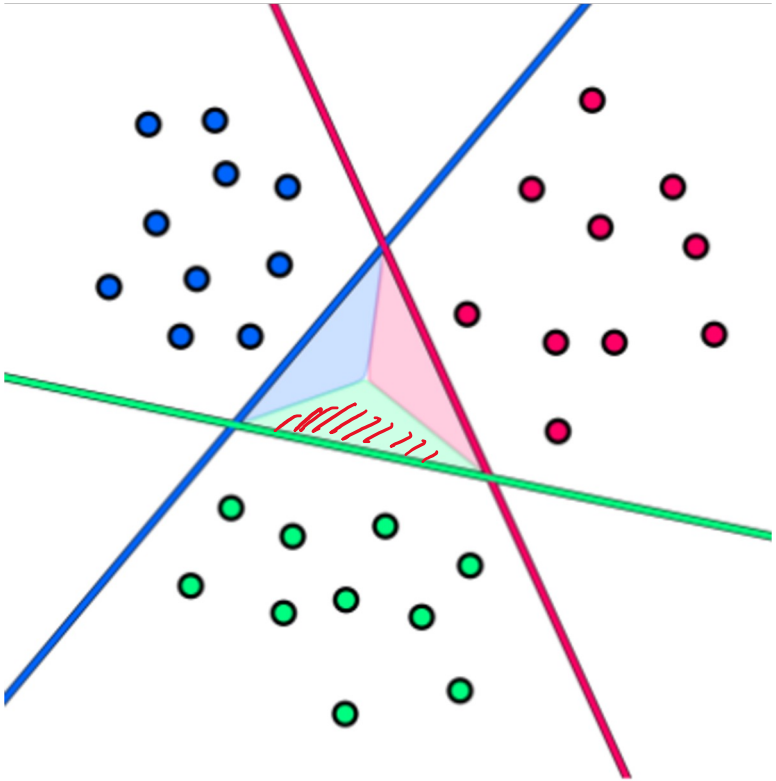
$$y = \underset{c=0,\ldots,C-1}{\text{argmax}} \quad \dot{\mathbf{x}}^T \mathbf{w}_c.$$

Points on the negative side of all classifiers



In this case we cannot argue that one classifier is more 'confident' in the class identity of such points. However, we can ask which classifier is the least 'unsure' about the class identity of such points?

If we repeat this logic for every point in the region ...



Classification rule: we assign a point to the class whose boundary is at the largest signed distance from it.

Expressed algebraically, once again we have

$$y = \underset{c=0,\ldots,C-1}{\operatorname{argmax}} \; \dot{\mathbf{x}}^T \mathbf{w}_c .$$

Putting it all together

- We have now deduced that the single rule for assigning a label ‘y’ to a point $\mathbf{x}$

$$y = \underset{c=0,\ldots,C-1}{\operatorname{argmax}} \; \dot{\mathbf{x}}^T \mathbf{w}_c.$$

- which assigns the label based on the **maximum signed distance of this point to each classifier**, applies to the entire space of our problem.
- We call this the **fusion rule** since it tells us precisely how to fuse our C individual classifiers together to make a unified and consistent classification across the entire space of any dataset.

One-versus-All multi-class classification

1: **Input:** multiclass dataset $\left\{\left(\mathbf{x}_p,\, y_p\right)\right\}_{p=1}^{P}$ where $y_p \in \{0, \ldots, C-1\}$, two-class classification scheme and optimizer

2: `for` $j = 0, \ldots, C-1$

3: form temporary labels $\tilde{y}_p = \begin{cases} +1 & \text{if } y_p = j \\ -1 & \text{if } y_p \neq j \end{cases}$

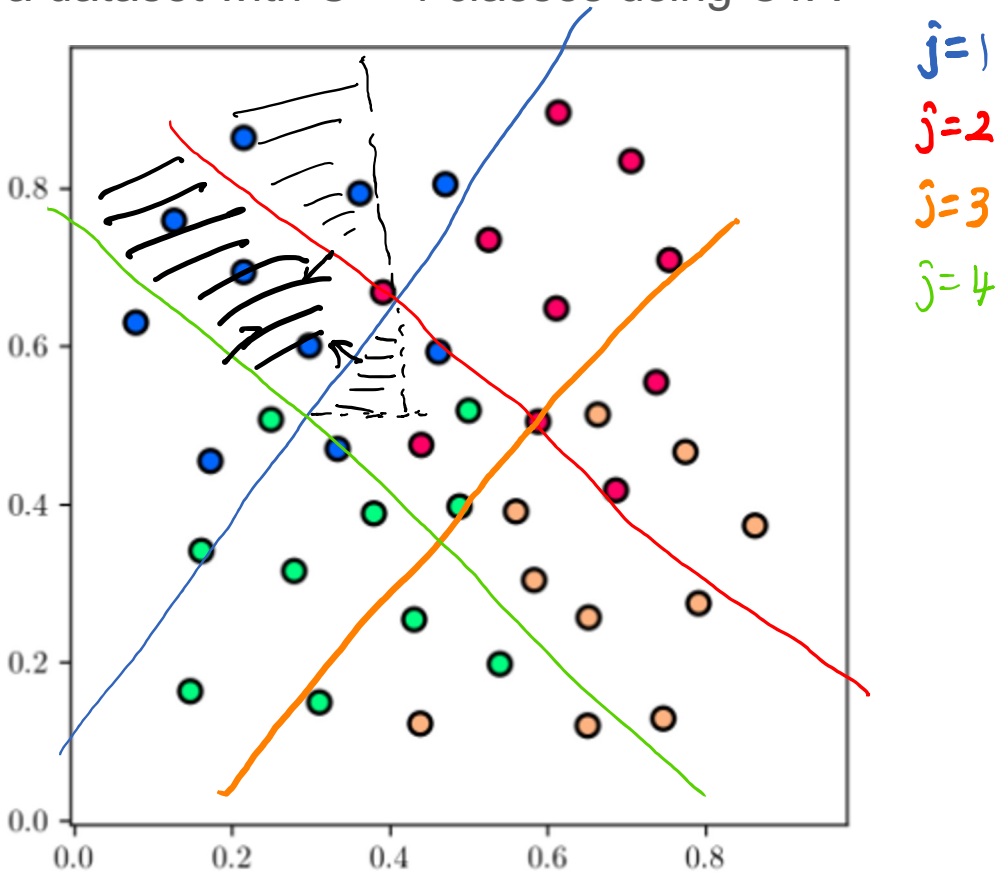
4: solve two-class subproblem on $\left\{\left(\mathbf{x}_p,\, \tilde{y}_p\right)\right\}_{p=1}^{P}$ to find weights $\mathbf{w}_j$

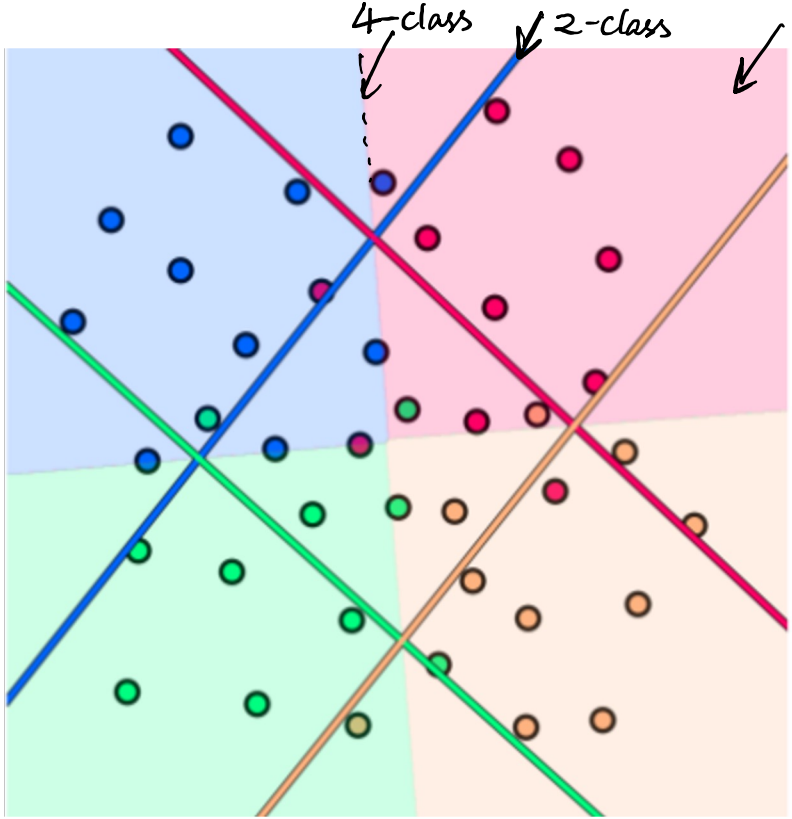
5: normalize classifier weights by magnitude of feature-touching portion $\mathbf{w}_j \longleftarrow \dfrac{\mathbf{w}_j}{\|\boldsymbol{\omega}_j\|_2}$

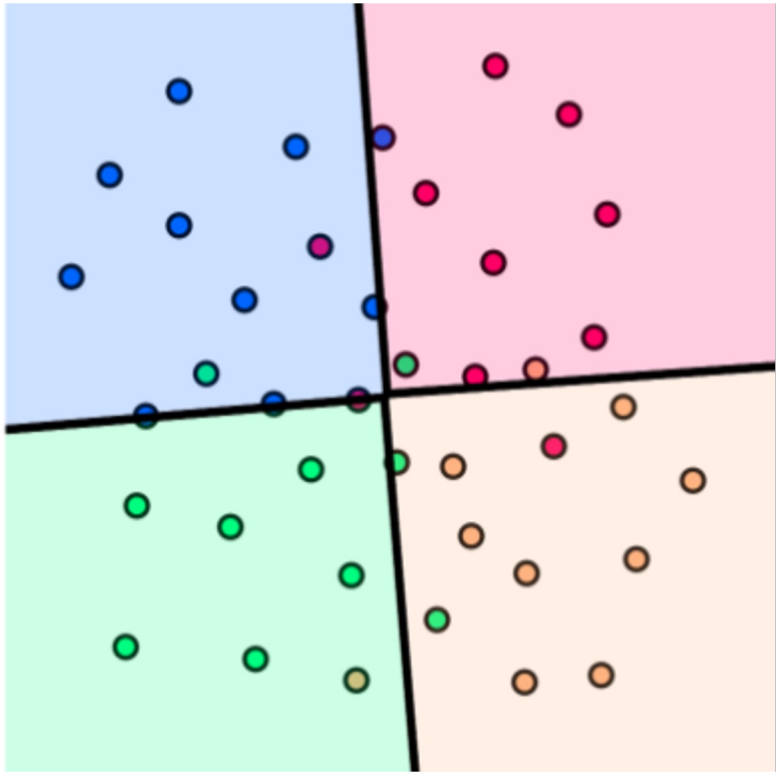
6: `end for`

7: To assign label y to a point $\mathbf{x}$, apply the fusion rule: $y = \underset{c=0,\ldots,C-1}{\operatorname{argmax}} \; \mathring{\mathbf{x}}^T \mathbf{w}_c$

Example: Classifying a dataset with C = 4 classes using OvA







Note with this dataset that each class is *not* linearly separable from the remainder of the data. This is no matter - the OvA framework still produces an appropriate multi-class boundary.

7.3 Multi-class classification and the Perceptron

Once again we deal with an arbitrary multi-class dataset $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$ consisting of C distinct classes of data.

The labels for these classes can be made arbitrarily, but here we will once again employ label values

$$y_p \in \{0, 1, \ldots, C-1\}$$

Recall the fusion rule

$$y_p = \underset{c=0,\ldots,C-1}{\operatorname{argmax}} \; \dot{\mathbf{x}}_p^T \mathbf{w}_c.$$

$$x_p^T w_{y_p} \approx \max_{c=0..C-1} x_p^T w_c$$

$$\Downarrow$$

$$\max_{c=0..C-1} x_p^T w_c - x_p^T w_{y_p} = 0 \quad \text{if } c = y_p$$

$$\text{otherwise} > 0$$

Our aim is (instead of tuning our each classifier's weights one-by-one and then combining them) to **learn all sets of weights simultaneously so as to satisfy this ideal condition as often as possible**.

Let's slightly re-write the fusion rule:

$$\dot{\mathbf{x}}_p^T \mathbf{w}_{y_p} = \max_{c=0,\ldots,C-1} \dot{\mathbf{x}}_p^T \mathbf{w}_c.$$

Subtracting $\dot{\mathbf{x}}_p^T \mathbf{w}_{y_p}$ from both sides gives a point-wise cost that is *always nonnegative* and minimal at zero

$y_p \in \{0, 1, 2, \ldots C-1\}$

$$g_p\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right) = \left(\max_{c=0,\ldots,C-1} \dot{\mathbf{x}}_p^T \mathbf{w}_c\right) - \dot{\mathbf{x}}_p^T \mathbf{w}_{y_p}.$$

predicted — ground truth

- Taking the *average* of this point-wise cost over the entire dataset we have

$$g\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right) = \frac{1}{P}\sum_{p=1}^{P}\left[\left(\max_{c=0,\ldots,C-1} \dot{\mathbf{x}}_p^T\mathbf{w}_c\right) - \dot{\mathbf{x}}_p^T\mathbf{w}_{y_p}\right]$$

  gives us the **multi-class Perceptron** cost function

- The function is **convex**
- It has a **trivial solution at zero** (which is often avoided by initializing local optimization away from the origin).

The **multi-class Perceptron cost** is a direct generalization of its two-class version introduced earlier.

- Two-class perceptron cost: $$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} \max\left(0, -y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right).$$

- The multi-class Perceptron cost in the following equivalent form

$$g\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right) = \frac{1}{P} \sum_{p=1}^{P} \max_{\substack{c=0,\ldots,C-1 \\ c \neq y_p}} \left(0, \dot{\mathbf{x}}_p^T \left(\mathbf{w}_c - \mathbf{w}_{y_p}\right)\right).$$

max(a, b) – c = max(a-c, b-c)

$$g\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right) = \frac{1}{P} \sum_{p=1}^{P} \left[\left(\max_{c=0,\ldots,C-1} \dot{\mathbf{x}}_p^T \mathbf{w}_c\right) - \dot{\mathbf{x}}_p^T \mathbf{w}_{y_p}\right]$$

$$g_p(w_0, w_1, \dots, w_{C-1}) = \max_{c=0,\dots C-1} (x_p^T w_c) - x_p^T w_{y_p}$$

$$g(w) = \max\{0, -y x_p^T w\}$$
$$y \in \{-1, 1\}$$

$y_p \in \{0 \dots C-1\}$

$$= \max\{x_p^T w_0 - x_p^T w_{y_p}, x_p^T w_1 - x_p^T w_{y_p}, \dots, x_p^T w_{C-1} - x_p^T w_{y_p}\}$$

$$= \max_{\substack{c=0\dots C-1, \\ c \neq y_p}} \{0, x_p^T w_c - x_p^T w_{y_p}\}$$

$C = 2 \Rightarrow$

$$g_p(w_0, w_1) = \max_{\substack{c=0,1 \\ c \neq y_p}} \{0, x_p^T w_c - x_p^T w_{y_p}\}$$

if $y_p = 1 \Rightarrow$ $= \max\{0, x_p^T w_0 - x_p^T w_1\} = \max\{0, x_p^T (w_0 - w_1)\}$

else $y_p = 0 \Rightarrow$ $= \max\{0, x_p^T w_1 - x_p^T w_0\} = \max\{0, x_p^T (w_1 - w_0)\}$

$\max\{0, x_p^T w'\} \Leftrightarrow y = 1$

w'

$\max\{0, -x_p^T w'\} \Leftrightarrow y = -1$

$\max\{0, -y x_p^T (-w')\}$

Regularization and the multi-class Perceptron

Recall that in order to fairly compare the distance of each input $\mathbf{x}_p$ to our two-class decision boundaries, we should formally subject the cost to the constraints that all normal vectors have unit length.

Separating the biases from feature-touching weights, this leads to the following constrained problem

$$\underset{b_0,\,\boldsymbol{\omega}_0,\,\ldots,\,b_{C-1},\,\boldsymbol{\omega}_{C-1}}{\text{minimize}} \quad \frac{1}{P}\sum_{p=1}^{P}\left[\left(\max_{c\,=\,0,\ldots,C-1} b_c + \mathbf{x}_p^T\boldsymbol{\omega}_c\right) - \left(b_{y_p} + \mathbf{x}_p^T\boldsymbol{\omega}_{y_p}\right)\right]$$

$$\text{subject to} \quad \left\|\boldsymbol{\omega}_c\right\|_2^2 = 1, \quad c = 0,\ldots,C-1$$

Approximately solve it by *relaxing the constraints*

$$\frac{1}{P}\sum_{p=1}^{P}\left[\left(\max_{c=0,\ldots,C-1} b_c + \mathbf{x}_p^T\boldsymbol{\omega}_c\right) - \left(b_{y_p} + \mathbf{x}_p^T\boldsymbol{\omega}_{y_p}\right)\right] + \lambda\sum_{c=0}^{C-1}\|\boldsymbol{\omega}_c\|_2^2$$

- For simplicity we choose a single regularization parameter $\lambda \geq 0$ that is used to penalize the magnitude of all normal vectors simultaneously.
- Even though this regularized form will not necessarily guarantee that $\|\boldsymbol{\omega}_c\|_2^2 = 1$ for all ‘c’, it will generally force the length of all normal vectors to behave well, e.g., disallowing one normal vector to grow arbitrarily large while one shrinks to almost nothing.

The Multi-class Softmax / Cross Entropy cost function

As was the case with the two-class perceptron, here too we can *smooth* the multi-class Perceptron cost employing the **softmax** function, defined as

$$\text{soft}\left(s_0, s_1, \ldots, s_{C-1}\right) = \log\left(e^{s_0} + e^{s_1} + \cdots + e^{s_C\ 1}\right)$$

which is a close and smooth approximation to the **max** function

$$\text{soft}\left(s_0, s_1, \ldots, s_{C-1}\right) \approx \max\left(s_0, s_1, \ldots, s_{C-1}\right).$$

Replacing the $\max$ function in each summand of the multi-class Perceptron gives

$$g\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right) = \frac{1}{P}\sum_{p=1}^{P}\left[\log\left(\sum_{c=0}^{C-1} e^{\dot{\mathbf{x}}_p^T\mathbf{w}_c}\right) - \dot{\mathbf{x}}_p^T\mathbf{w}_{y_p}\right].$$

This is referred to as the **multi-class Softmax** cost function, which

- is convex
- has infinitely many smooth derivatives
- no longer has a trivial solution at zero

This cost function goes by *many* names in practice including: the **multi-class Softmax**, **Multiclass Cross Entropy**, **Softplus**, and **Multiclass Logistic** cost to name a few.

$$g\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right) = \frac{1}{P}\sum_{p=1}^{P}\left[\log\left(\sum_{c=0}^{C-1} e^{\dot{\mathbf{x}}_p^T \mathbf{w}_c}\right) - \dot{\mathbf{x}}_p^T \mathbf{w}_{y_p}\right].$$

Why multi-class Softmax?

Because it is a softmax-smoothed version of the Multiclass Perceptron and because it is the natural generalization of the two-class version

Two-class softmax cost: $$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right)$$

Same when C=2 and y_p = -1/1

Multi-class version can be rewritten as:

$$g\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + \sum_{\substack{c=0 \\ j\neq y_p}}^{C-1} e^{\dot{\mathbf{x}}_p^T\left(\mathbf{w}_c - \mathbf{w}_{y_p}\right)}\right).$$

Hint: x = log(exp(x))

Why multi-class Cross Entropy?

Because - likewise - it is a natural generalization of the two class-version

$$g(\mathbf{w}) = -\frac{1}{P}\sum_{p=1}^{P} y_p \log\left(\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) + \left(1 - y_p\right)\log\left(1 - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right).$$

Multi-class version can be rewritten as:

Same when y_p=0/1 and C=2

$$g\left(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}\right) = -\frac{1}{P}\sum_{p=1}^{P} \log\left(\frac{e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_{y_p}}}{\sum_{c=0}^{C-1} e^{\mathring{\mathbf{x}}_p^T \mathbf{w}_c}}\right).$$

Regularization and the multi-class Softmax

As with the multi-class Perceptron, it is common to *regularize* the Multiclass Softmax via its feature touching weights as

$$\frac{1}{P}\sum_{p=1}^{P}\left[\log\left(\sum_{c=0}^{C-1}e^{b_c+\mathbf{x}_p^T\boldsymbol{\omega}_c}\right)-\left(b_{y_p}+\mathbf{x}_p^T\boldsymbol{\omega}_{y_p}\right)\right]+\lambda\sum_{c=0}^{C-1}\|\boldsymbol{\omega}_c\|_2^2$$

Implementing and minimizing a modular multi-class softmax in `Python`

- Stack weights into a matrix format:

$$\mathbf{W} = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & \cdots & w_{0,C-1} \\ w_{1,0} & w_{1,1} & w_{1,2} & \cdots & w_{1,C-1} \\ w_{2,0} & w_{2,1} & w_{2,2} & \cdots & w_{2,C-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ w_{N,0} & w_{N,1} & w_{N,2} & \cdots & w_{N,C-1} \end{bmatrix}.$$

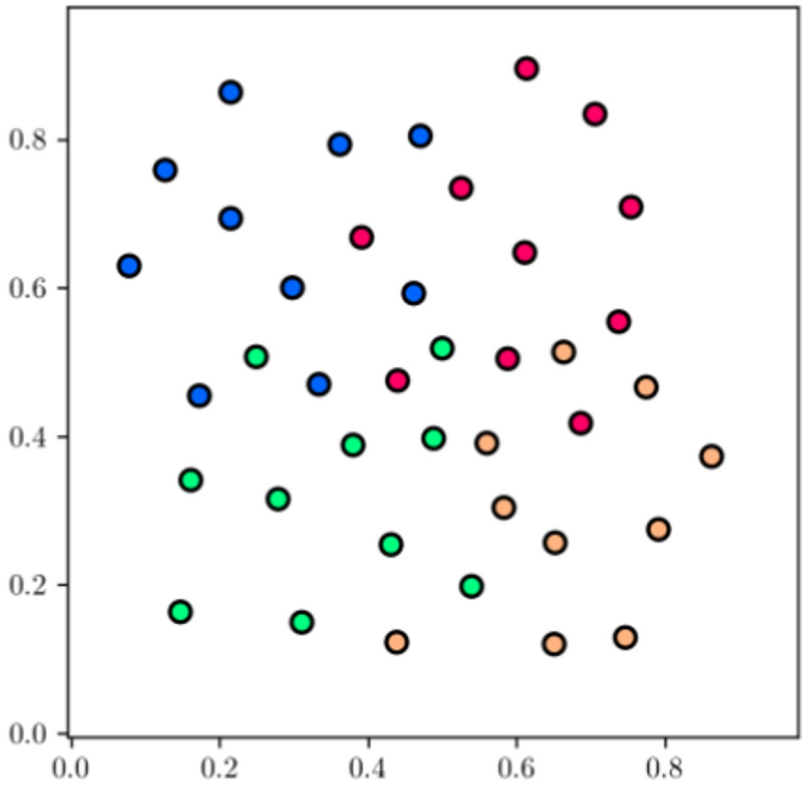
- Compute the linear regressor output for all C classes

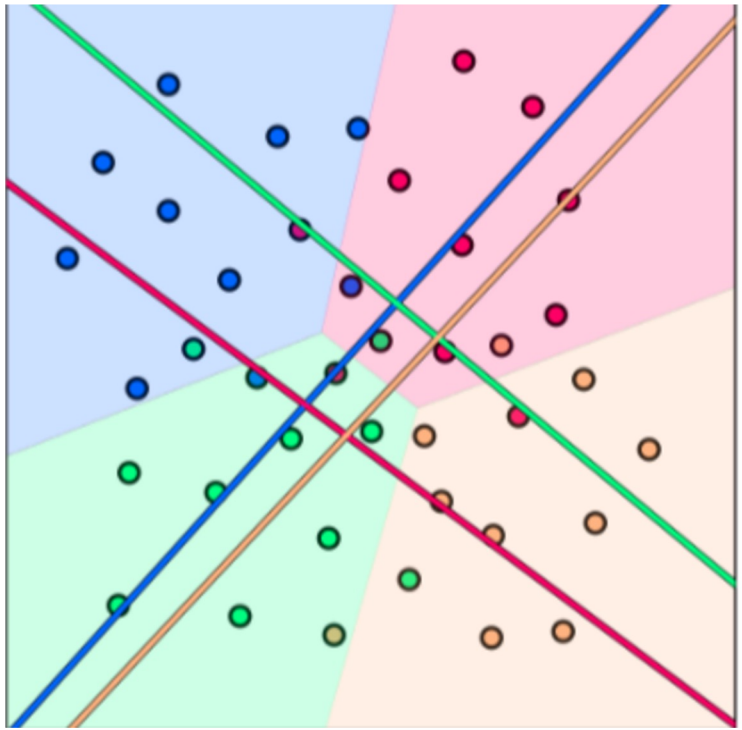
```
# compute C linear combinations of input point, one per classifier
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

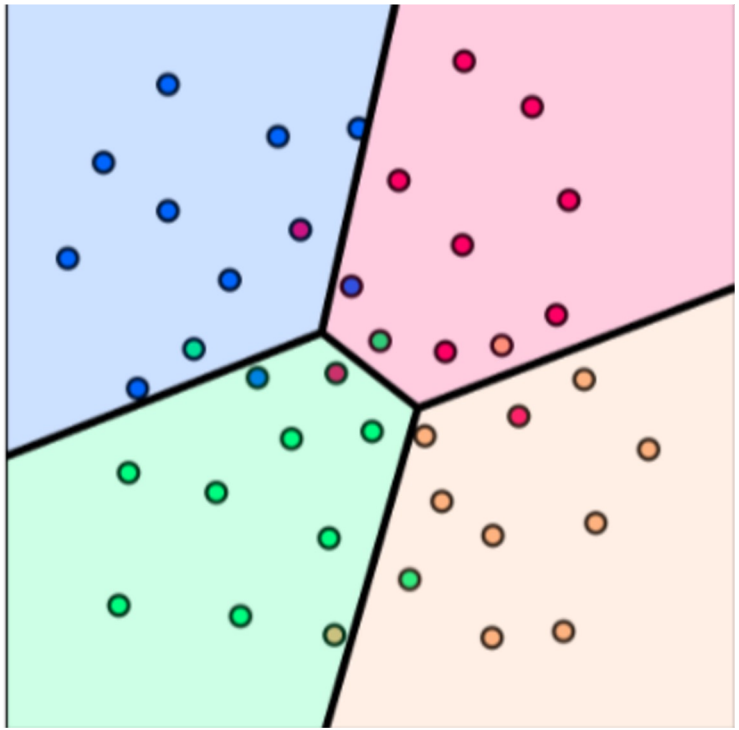
```python
# multiclass perceptron
lam = 10**-5  # our regularization parameter
def multiclass_perceptron(w):
    # pre-compute predictions on all points
    all_evals = model(x,w)

    # compute maximum across data points
    a = np.max(all_evals,axis = 0)

    # compute cost in compact form using numpy broadcasting
    b = all_evals[y.astype(int).flatten(),np.arange(np.size(y))]
    cost = np.sum(a - b)

    # add regularizer
    cost = cost + lam*np.linalg.norm(w[1:,:],'fro')**2

    # return average
    return cost/float(np.size(y))
```

```
# multiclass softmaax regularized by the summed length of all normal vectors
lam = 10**(-5)  # our regularization paramter
def multiclass_softmax(w):
    # pre-compute predictions on all points
    all_evals = model(x,w)

    # compute softmax across data points
    a = np.log(np.sum(np.exp(all_evals),axis = 0))

    # compute cost in compact form using numpy broadcasting
    b = all_evals[y.astype(int).flatten(),np.arange(np.size(y))]
    cost = np.sum(a - b)

    # add regularizer
    cost = cost + lam*np.linalg.norm(w[1:,:],'fro')**2

    # return average
    return cost/float(np.size(y))
```

Example: Multi-class Softmax on a toy dataset with C = 4 classes







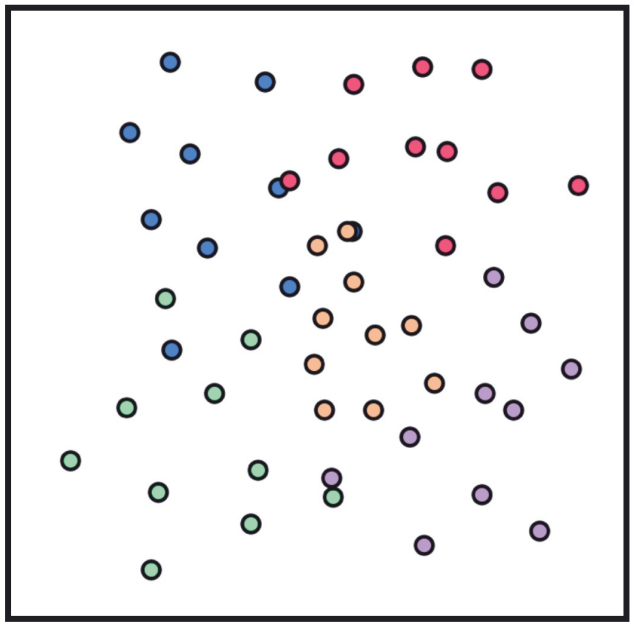
7.4 Which approach produces the best results?

- We have now seen two fundamental approaches to linear multi-class classification: the One-versus-All (OvA) and the Multi-class Perceptron / Softmax.
- Both approaches are commonly used in practice and can - depending on the dataset - produce similarly good results.
- However the latter approach (the Multi-class Perceptron / Softmax) is - at least in principle - capable of achieving higher accuracy on a broader range of datasets.

- This is due to the fact that with OvA we solve a sequence of C two-class subproblems (one per class), tuning the weights of our classifiers independently of each other.
- Only afterward do we combine all classifiers together to form the fusion rule.
- Thus the weights we learn satisfy the fusion rule indirectly.

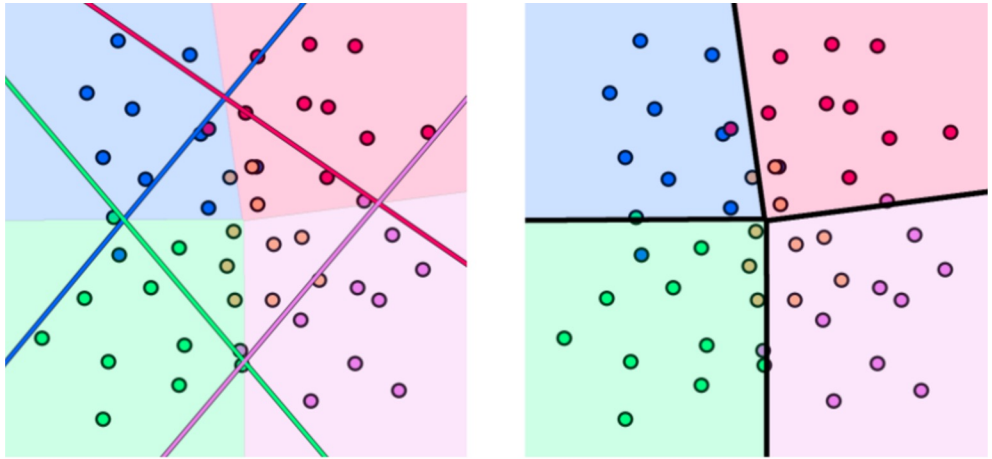
- On the other hand, with the multi-class Perceptron or Softmax cost function minimization we are tuning all parameters of all C classifiers *__simultaneously__*...
- ...to directly satisfy the fusion rule over our training dataset.
- This joint minimization permits potentially valuable interactions to take place in-between the two-class classifiers in the tuning of their weights that cannot take place in the OvA approach.

We illustrate this principal superiority of the Multi-class Perceptron / Softmax approach over OvA using a toy C = 5 class dataset shown below. Here points colored red, blue, green, khaki, and violet have label values $y_p = 0$, 1, 2 , 3, and 4 respectively.

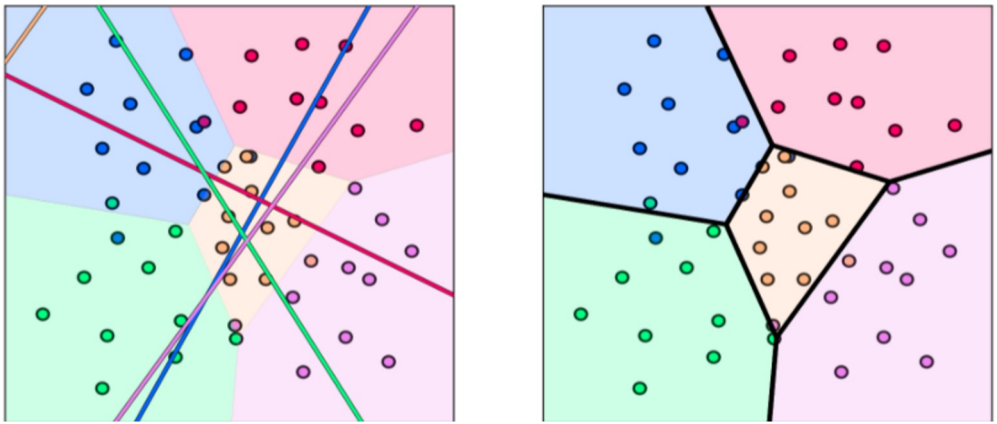


- Think for a moment how the OvA approach will perform *in terms of the khaki colored class*.
- In particular, think of how the subproblem in which we distinguish between members of this class and all others will be solved.
- Because this class of data is *surrounded* by members of the other classes - and there fewer members of the khaki class then all other classes combined - the *optimal* classification rule for this subproblem is to *classify all points as non-khaki* (or in other words to *misclassify* the entire khaki class).

- Now because the weights of decision boundary associated with the khaki colored class are tuned *solely based on this subproblem*, this will lead to the *entire khaki class being misclassified in the final OvA solution*.
- This is shown below in the right panel of the figure below, along with the fused decision boundary.



- If - on the other hand - we employ the Multi-class Perceptron or Softmax approach we will not miss this class.
- Here - because we tune all weights ***together*** in order to minimize the number of misclassifications over the entire dataset - we end with final fused decision boundary that is far superior to the one provided by OvA (shown in the right panel below).



- We misclassify far fewer points and - in particular - do *not* misclassify the entire khaki class of data.
- While the decision boundary associated with the khaki class lies outside the range of the datapoints, because we have tuned everything together weights in other individual decision boundaries are *adjusted to better satisfy our goal of minimizing misclassifications*.

7.6 Classification Quality Metrics

- Assume the default label values $y_p \in \{0, 1, \ldots, C-1\}$
- We denote the optimal set of weights found by minimizing a classification cost function (or via performing OvA) by $\mathbf{w}^\star$
- Then note we can write our fully tuned linear model as

$$\text{model}\left(\mathbf{x}, \mathbf{W}^\star\right) = \dot{\mathbf{x}}^T \mathbf{W}^\star$$

- This fully trained model defines an optimal decision boundary for the training dataset which is defined by the fusion rule

$$\underset{c=0,\ldots,C-1}{\operatorname{argmax}} \quad \mathbf{x}^T \mathbf{w}_c^\star .$$

- To predict the label y of an input $\mathbf{x}$ we then process it through this rule as

$$y \;=\; \underset{c=0,\ldots,C-1}{\operatorname{argmax}} \quad \mathbf{x}^T \mathbf{w}_c^\star$$

Judging the quality of a trained model using *accuracy*

- To *count the number of misclassifications* a trained multi-class classifier forms over our training dataset we simply take a raw count of the number of training datapoints $\mathbf{x}_p$ whose true label y_p is predicted *incorrectly*.
- To compare the point $\mathbf{x}_p$'s predicted label

$$\hat{y}_p = \underset{j=0,...,C-1}{\text{argmax}} \ \mathbf{x}_p^T \mathbf{w}_j^\star$$

and true true label y_p we can use an identity function $\mathcal{I}(\cdot)$ and compute

$$\mathcal{I}\left(\hat{y}_p, y_p\right) = \begin{cases} 0 & \text{if } \hat{y}_p, = y_p \\ 1 & \text{if } \hat{y}_p, \neq y_p. \end{cases}$$

- Summing all P points gives the total number of misclassifications of our trained model

$$\text{number of misclassifications} = \sum_{p=1}^{P} \mathcal{I}\left(\hat{y}_p,\, y_p\right).$$

- Using this we can also compute the *accuracy* - denoted $\mathcal{A}$ - of a trained model.
- This is simply the percentage of training dataset whose labels are correctly predicted by the model.

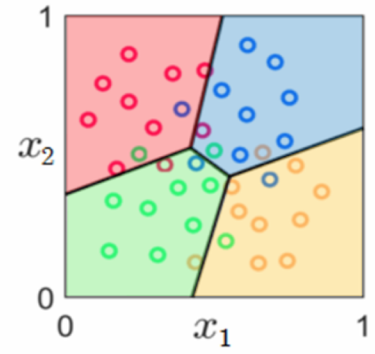
$$\mathcal{A} = 1 - \frac{1}{P}\sum_{p=1}^{P} \mathcal{I}\left(\hat{y}_p,\, y_p\right).$$

- The accuracy ranges from 0 (no points are classified correctly) to 1 (all points are classified correctly).

The confusion matrix and additional quality metrics

We can generalize the idea of a confusion matrix, introduced for two-class classification problems.

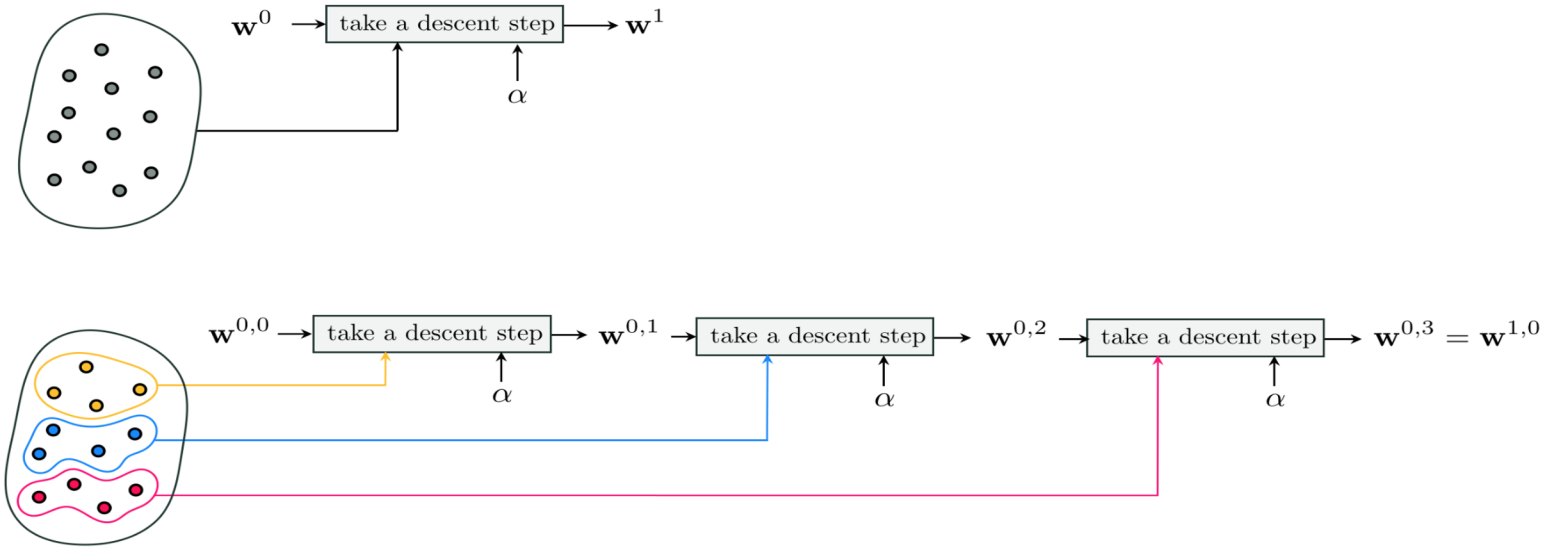
To form such matrices for problems with arbitrary number of classes. In general the confusion matrix associated to a classification problem with $C \geq 2$ classes will be of size $C \times C$, whose $(i,j)^{\text{th}}$ entry is the number of data points that have $y = i$ and $\hat{y} = j$.



$$\begin{array}{c} \\ \text{red} \\ \text{blue} \\ \text{green} \\ \text{yellow} \end{array} \begin{array}{c} \begin{array}{cccc} \text{red} & \text{blue} & \text{green} & \text{yellow} \end{array} \\ \begin{pmatrix} 8 & 1 & 1 & 0 \\ 1 & 7 & 1 & 1 \\ 1 & 1 & 7 & 1 \\ 0 & 1 & 1 & 8 \end{pmatrix} \end{array}$$

7.7 Stochastic and Mini-Batch Learning

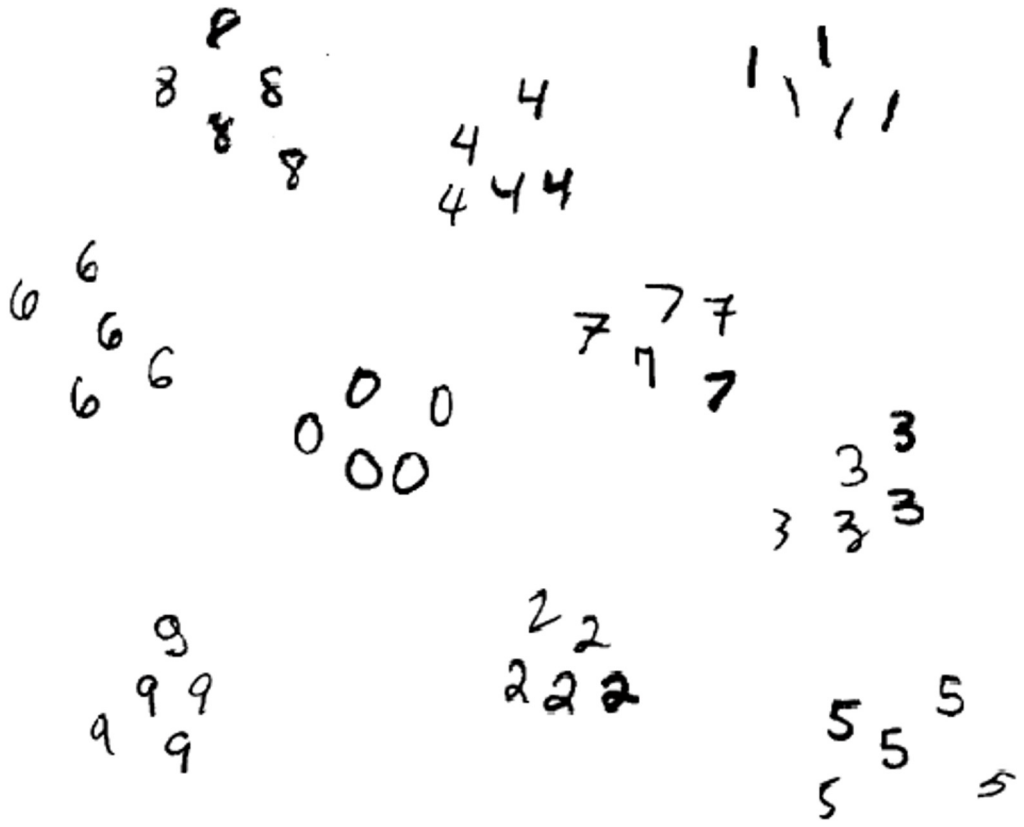
- As opposed to a standard (also called full batch) local optimization technique that takes individual steps by sweeping through an *entire* set of training data *simultaneously*, the mini-batch approach has us take smaller steps sweeping through training data *sequentially*.
- One complete sweep through the data being referred to as an *epoch* of mini-batch learning.



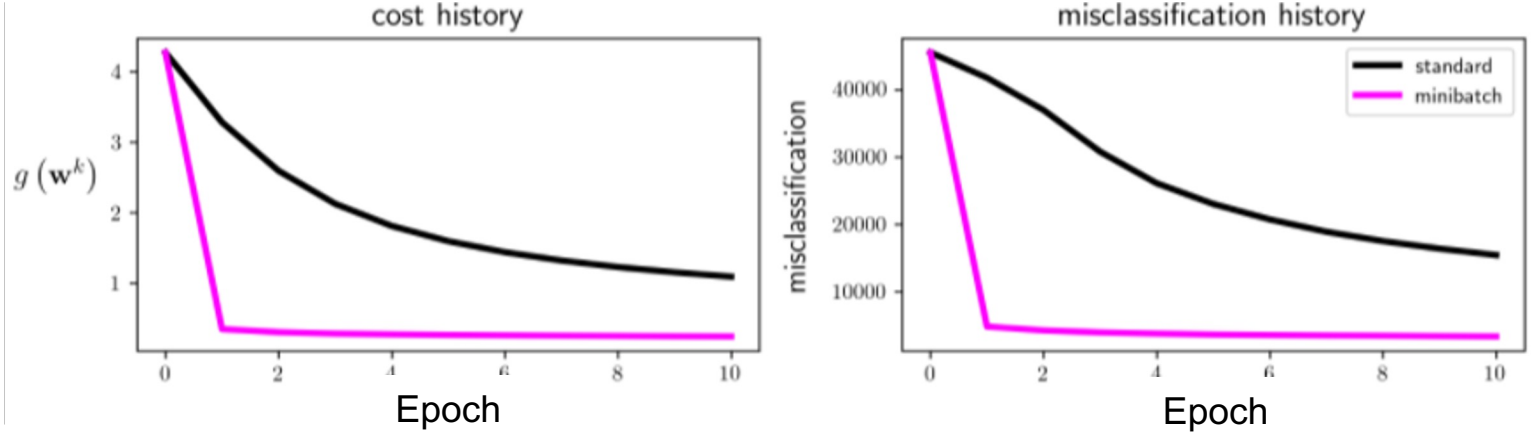
- **Mini-batch learning often greatly accelerates the minimization of machine learning cost functions** (and thus the corresponding learning taking place).
- This is particularly true when dealing with ***very large datasets***, i.e., when P is large.
- With very large datasets the mini-batch approach can also help limit the amount of active memory consumed in storing data by loading in - at each step in a mini-batch epoch - *only the data included in the current mini-batch*.

Example: Comparing standard to minibatch gradient descent

- Recognizing handwritten digits is a popular multiclass classification problem commonly built into the software of mobile banking applications.
- e.g., the ability to automatically deposit paper checks. Here each class of data consists of (images of) several handwritten version of a single digit in the range 0-9, giving a total of 10 classes.



- Mini-batch gradient descent vs. the Softmax cost over the standard method using. $P = 50,000$ random training points from the MNIST dataset.
- In particular we show a comparison of the first 10 steps / epochs of both methods, using a batch of size 200 for the mini-batch size and the same steplength for both runs
- Here we see that the mini-batch run drastically accelerates minimization in terms of both the cost function (left panel) and number of misclassifications (right panel).



Fun Stuff

How does batch size affect training?

Batch size is one of the hyperparameters in deep learning and can significantly affect both the training dynamics and the quality of the resulting model. Here's a breakdown of how batch size affects training:

1. **Convergence Speed**:
   - **Smaller Batches**: Typically, smaller batches result in more frequent weight updates in a given epoch, which can lead to faster convergence. However, this doesn't mean the overall training time is shorter since you might need more epochs to achieve convergence.
   - **Larger Batches**: Larger batches provide a more accurate estimate of the gradient. However, they might converge in fewer epochs but require more time per epoch.
2. **Generalization**:
   - **Smaller Batches**: There is some evidence suggesting models trained with smaller batches generalize better. This could be because the noisier gradients with small batch training act as a form of regularization.
   - **Larger Batches**: Models trained with large batches might have the risk of overfitting, especially if the model does not see enough unique data points.
3. **Memory Utilization**:
   - **Smaller Batches**: Less memory-intensive and can be used on hardware with limited memory.
   - **Larger Batches**: Require more memory. If the batch size is too large for the GPU memory, it can lead to out-of-memory errors.

4. **Stability of Training**:
    - **Smaller Batches**: The gradients can be noisy due to fewer samples, which can lead to instability in training. Techniques like gradient clipping or using adaptive learning rates can help.
    - **Larger Batches**: The gradients are averaged over more samples, leading to smoother and more stable training. But it may get stuck in sharp, non-generalizing minima.
5. **Parallel Processing**:
    - **Larger Batches**: Can benefit more from parallel processing capabilities of modern GPUs, leading to faster per-epoch training times.
6. **Learning Rate Sensitivity**:
    - **Smaller Batches**: Typically require a smaller learning rate since the gradients can be noisy.
    - **Larger Batches**: Can often use a larger learning rate without diverging. However, learning rate warm-up strategies might be employed to stabilize training in the initial phase.

In practice, the choice of batch size often comes down to empirical tuning. Researchers and practitioners try different values and select the one that offers a good trade-off between training speed, memory usage, and model performance. Sometimes, other techniques like gradient accumulation are used to simulate the effect of larger batches without the memory overhead.

https://arxiv.org/pdf/1706.02677.pdf

Tech Report: Cited by 3469

Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Priya Goyal Piotr Dollár Ross Girshick Pieter Noordhuis
Lukasz Wesolowski Aapo Kyrola Andrew Tulloch Yangqing Jia Kaiming He

Facebook

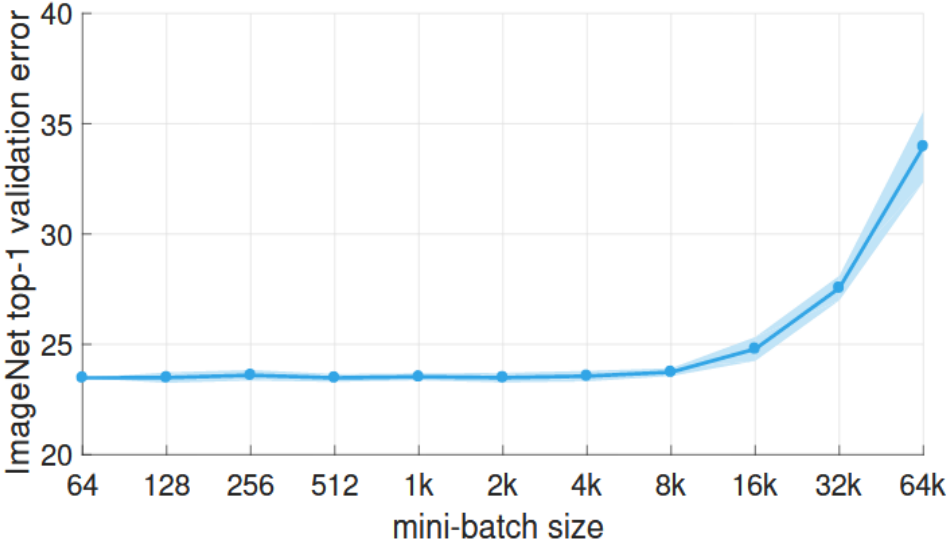


Figure 1. **ImageNet top-1 validation error *vs*. minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*. For all minibatch sizes we set the learning rate as a *linear* function of the minibatch size and apply a simple warmup phase for the first few epochs of training. All other hyper-parameters are kept fixed. Using this simple approach, accuracy of our models is invariant to minibatch size (up to an 8k minibatch size). Our techniques enable a linear reduction in training time with ~90% efficiency as we scale to large minibatch sizes, allowing us to train an accurate 8k minibatch ResNet-50 model in 1 hour on 256 GPUs.

Summary

- One vs. All
- Multi-class perceptron/cross entropy/softmax
- Multi-class metrics
  - Accuracy
  - Confusion matrix
- Mini-batch optimization