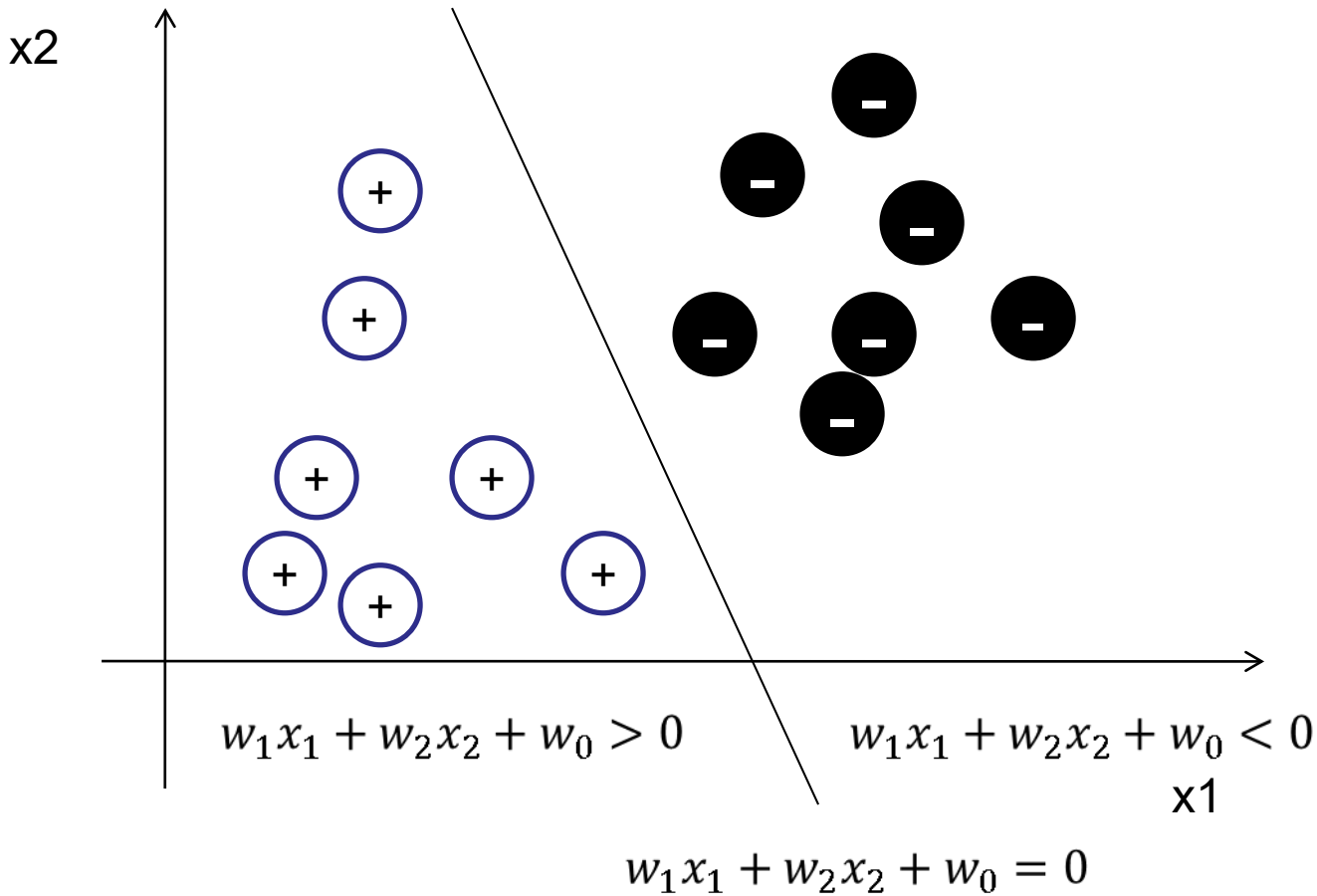
Linear classification models: Perceptron

Basic concepts:

The Perceptron algorithm
Perceptron loss/ hinge loss
Subgradient descent
Convergence proof of Perceptron
Concept of Margin
Voted and average Perceptrons
Structured Perceptron

Linear Classifier



Binary classification: General Setup

- Given a set of training examples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$, where each $\mathbf{x}_i \in R^{d+1}$, $y_i \in \{-1,1\}$

- Learn a linear function

$$g(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + \cdots + w_d x_d = \mathbf{w}^T \mathbf{x}$$

  Given an example $\mathbf{x} = [1, x_1, \ldots, x_d]^T$, we predict

  - $y(\mathbf{x}) = 1$ if $g(\mathbf{x}, \mathbf{w}) \geq 0$
  - $y(\mathbf{x}) = -1$ otherwise

- Goal: find a good $\mathbf{w}$ that minimizes some loss function $\mathcal{L}(\mathbf{w})$

Loss function

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} L(g(\mathbf{w}, \mathbf{x}_i), y_i)$$

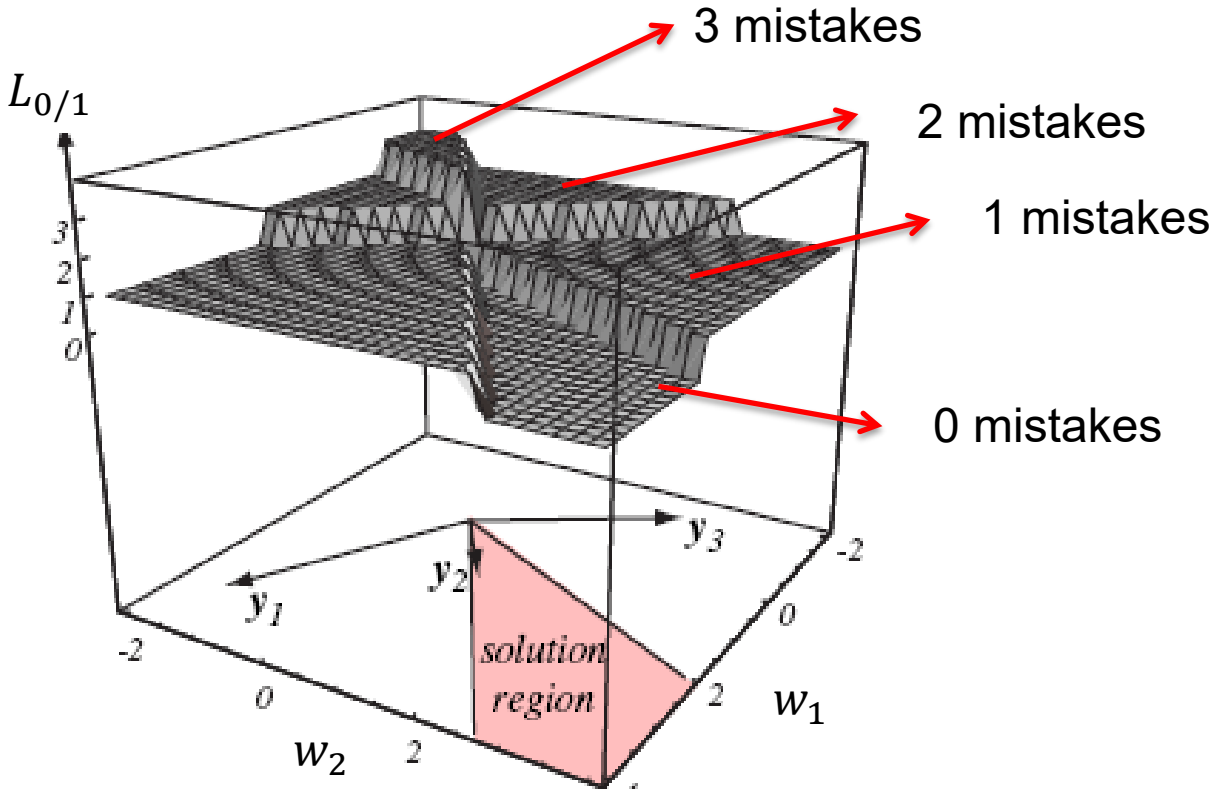
Where $L(g(\mathbf{w}, \mathbf{x}), y)$ is the loss of $g(\mathbf{w}, \mathbf{x})$ given its true label is y

0/1-loss:

$$L_{0/1}(g(\mathbf{w}, \mathbf{x}), y) = \begin{cases} 1 & \text{if } yg(\mathbf{w}, \mathbf{x}) < 0 \\ 0 & \text{otherwise} \end{cases}$$

This loss is conceptually aligned with our goal of maximizing accuracy, and minimizing error.

0/1 Loss counts the # of mistakes



Issue:

- Non convex – in general can be NP-hard to optimize
- Non-smooth – does not produce useful gradient since the surface of 0/1 loss is **piece-wise flat**

Perceptron Loss

$$L_p(g(\mathbf{w}, \mathbf{x}), y) = \max(0, -y\mathbf{w}^T\mathbf{x})$$

- If prediction is correct, $y\mathbf{w}^T\mathbf{x} > 0$,

$$L_p = \max(0, -y\mathbf{w}^T\mathbf{x}) = 0$$

- If prediction is incorrect, $y\mathbf{w}^T\mathbf{x} < 0$,

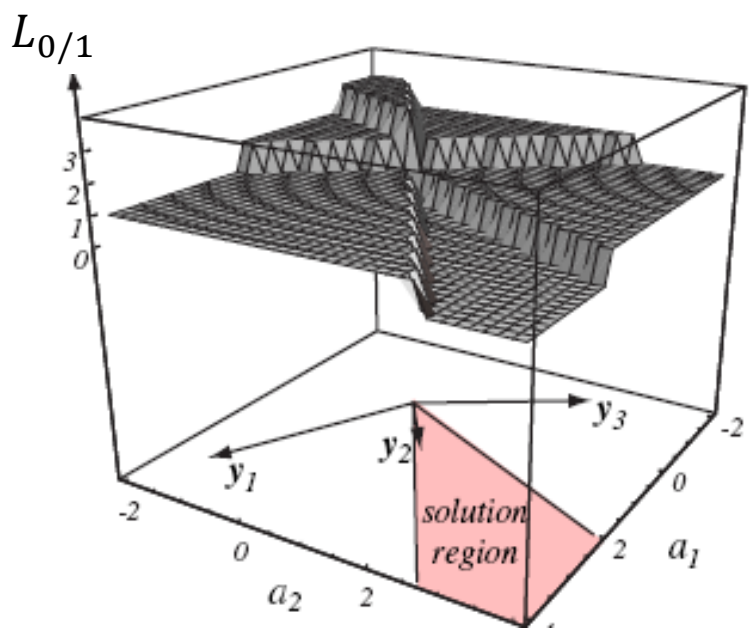
$$L_p = \max(0, -y\mathbf{w}^T\mathbf{x}) = -y\mathbf{w}^T\mathbf{x} > 0$$

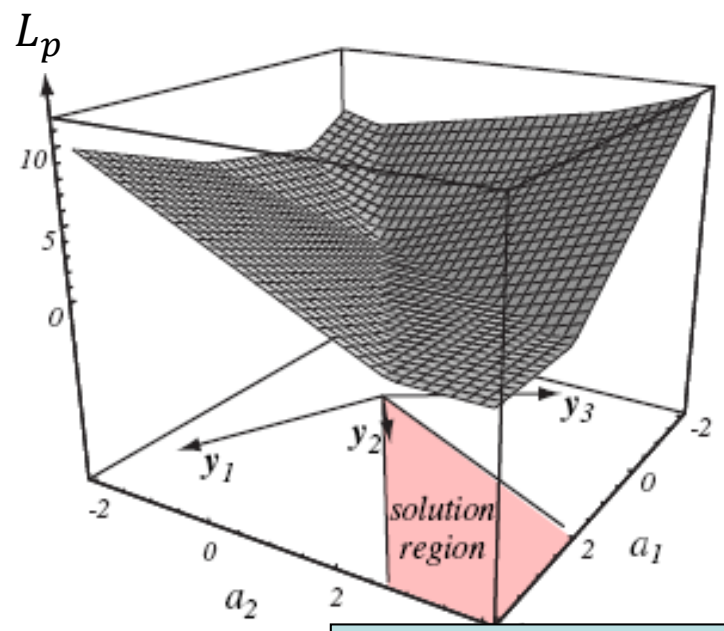
- When making a mistake, the loss is a linear function of $\mathbf{w}$

Perceptron Loss

$$L_p(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \max(0, -y_i \mathbf{w}^T \mathbf{x}_i)$$

- J_p is still non-smooth but piecewise linear
- Imagine if we drop a ball on this surface, it will follow gravity and goes to the solution region





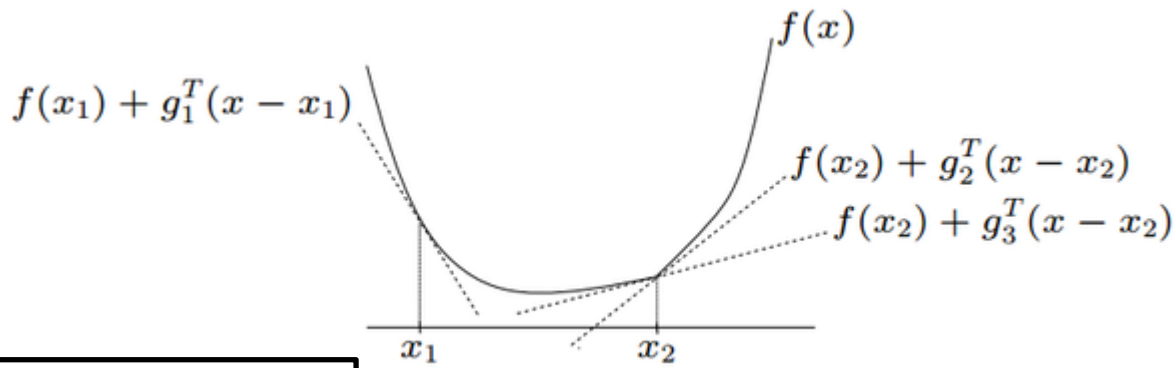
0/1 loss: piecewise constant

Perceptron criterion: piecewise linear

Subgradient (subderivative)

- The Subgradient/subderivative of a convex function is a way of generalizing the gradient/ derivative of a differentiable convex function at non-differentiable points
- g is a subgradient of f at x if, for all x', the following is true:

$$f(x') \geq f(x) + g^T(x' - x)$$



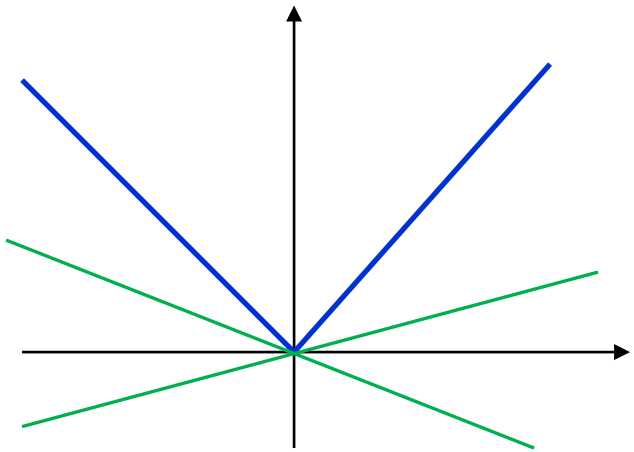
f is differentiable at x_1, subgradient g_1 is the gradient

f is not differentiable at x_2, g_2 and g_3 are subgradients at point x_2

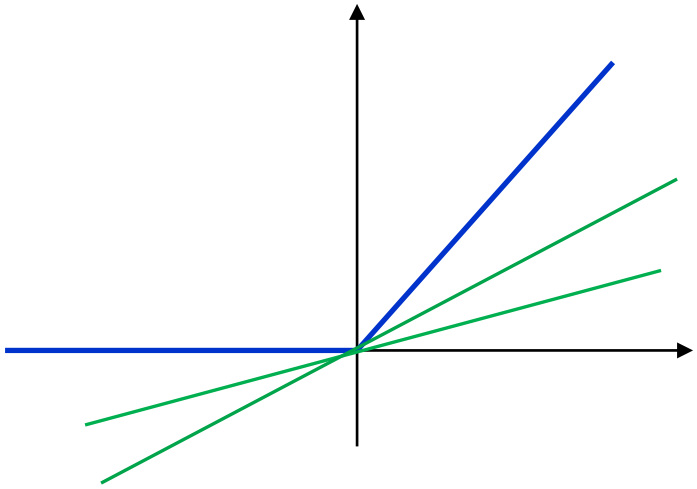
Image source: https://optimization.mccormick.northwestern.edu/index.php/Subgradient_optimization

Subgradient (subderivative) cont.

Example: $f(x) = |x|$
Where f is differentiable ($x\ != 0$), the sub-gradient = the gradient, i.g, $sign(x)$.
At x = 0, anything in the range [-1, 1] is a valid sub-gradient



For perceptron, the loss uses $f(x) = \max(0, x)$
f is differentiable when $x! = 0$: if $x > 0$, gradient is 1; if $x < 0$, the gradient is 0
At $x = 0$, anything in the range [0, 1] is a valid sub-gradient. Typically, we use 1.



Perceptron Update Rule (Online learning)

Perceptron Loss measured on example i:

$$L_i(\boldsymbol{w}) = \max(0, -y_i \boldsymbol{w}^T \boldsymbol{x}_i)$$

Subgradient contributed by example i:

$$\nabla L_i(\boldsymbol{w}) = \begin{cases} 0 & \text{if } y_i \boldsymbol{w}^T \boldsymbol{x}_i > 0 \\ -y_i \boldsymbol{x}_i & \text{otherwise} \end{cases}$$

y_i and $w^T x_i$ have the same sign, Correct prediction!

Perceptron Update Rule

After observing $(\mathbf{x}_i, y_i)$,

- if it is a mistake (i.e., $y_i w^T x_i \leq 0$) $\mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i$
- Otherwise do nothing

The (original) **Perceptron Algorithm**

(Stochastic gradient descent with constant step size)

$$
\begin{array}{l}
\text{Let } \mathbf{w} \leftarrow (0,0,0,\ldots,0) \\
\text{Repeat until convergence} \\
\qquad \text{for every training example } i = 1,\ldots,N: \\
\qquad\qquad \text{if } y_i \, \mathbf{w}^T \mathbf{x}_i \leq 0 \quad \mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i
\end{array}
$$

Online

- Look at one example at a time, update the model as soon as we make an error – as opposed to batch algorithms that update parameters after seeing the entire training set.

Error-driven

- We only update parameters/model if we make an error

Effect of the perceptron update

- Current weight $\mathbf{w}_t$ makes a mistaken on $(\mathbf{x}_i, y_i)$, i.e., $y_{\mathrm{i}}\mathbf{w}_t^T\mathbf{x}_i \leq 0$
- Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + y_i\mathbf{x}_i$$

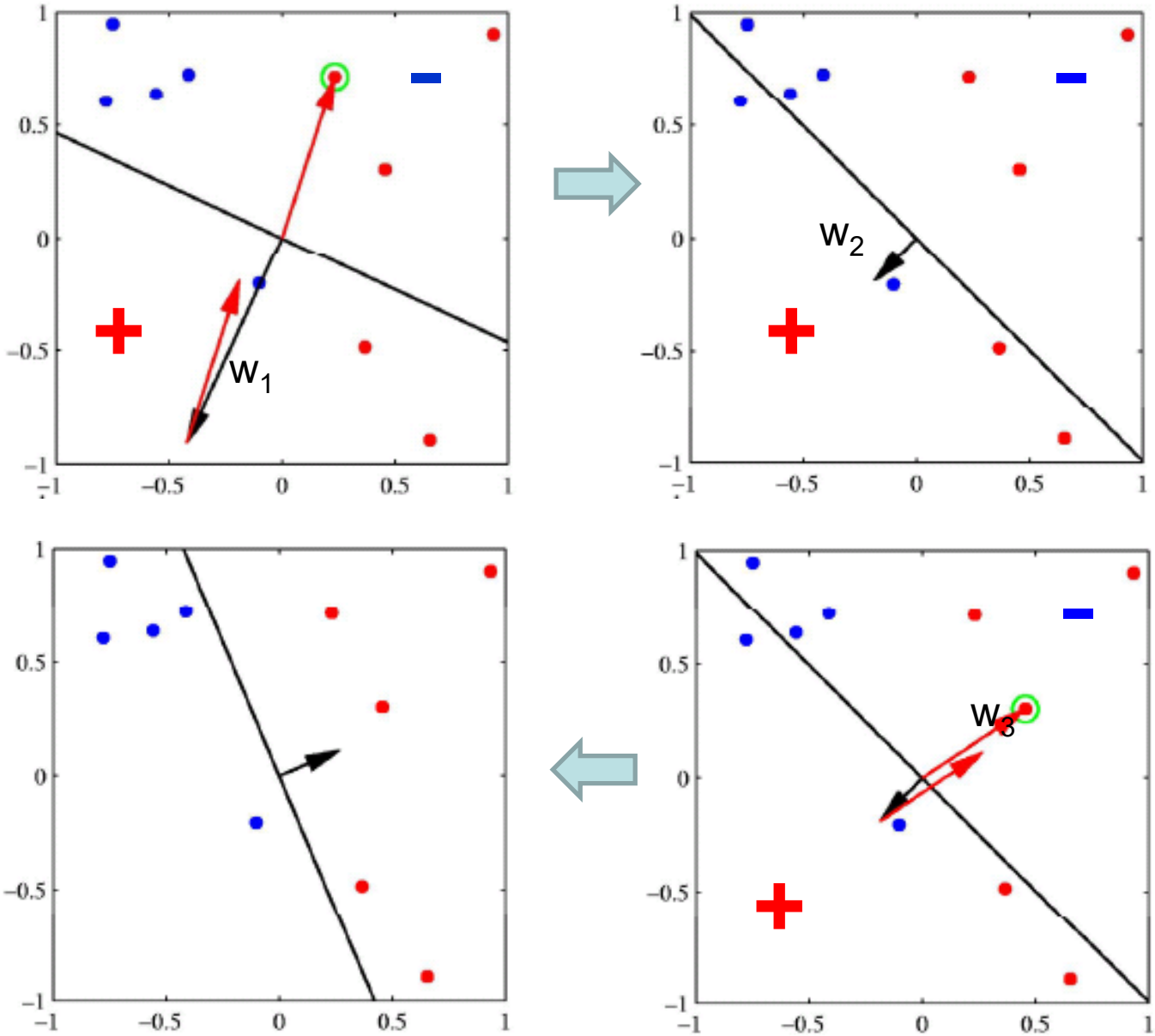
- Post update, we have:

$$y\mathbf{w}_{t+1}\mathbf{x}_i = y\mathbf{w}_t\mathbf{x}_i + y_i^2|x_i|^2$$
$$> y\mathbf{w}_t\mathbf{x}_i$$

- The update makes some correction for $(\mathbf{x}_i, y_i)$
  - But not guaranteed to be correct for $(\mathbf{x}_i, y_i)$

When an error is made, moves the weight in a direction that corrects the error

Update 1



Update 2

Convergence Theorem

(Block, 1962, Novikoff, 1962)

Given training example sequence $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots (\mathbf{x}_N, y_N)$.

If $\forall i,\ |\mathbf{x}_i| \le D$, and $\exists\, \mathbf{u},\ |\mathbf{u}| = 1$ and $y_i \mathbf{u}^T \mathbf{x}_i \ge \gamma > 0$ for all i,

then the number of mistakes that the perceptron algorithm makes is at most $(D/\gamma)^2$.

$|\cdot|$ denotes the Euclidean norm of a vector.

Proof

Let $\mathbf{u}$ be a unit vector that achieves γ margin, i.e., $|\mathbf{u}| = 1$ and $\forall i, y_i \mathbf{u}^T \mathbf{x}_i \geq \gamma$

Let $\boldsymbol{x}_k$ be the k-th mistake, we have $\boldsymbol{w}_k = \boldsymbol{w}_{k-1} + y_k \boldsymbol{x}_k$

Main idea: we want to show that the direction of $\mathbf{w}_k$ converges to $\mathbf{u}$,

i.e., $\frac{\mathbf{u}^{\mathrm{T}}\mathbf{w}_k}{|\mathbf{u}||\mathbf{w}_k|}$ converges to 1. To show this, we work out two parts:

1. $\mathbf{u}^T \mathbf{w}_k$ grows bigger as k increases
2. $|\mathbf{w}_k|$ does not grow as fast

Part 1:

$$\mathbf{u}^T \mathbf{w}_k = \mathbf{u}^T(\mathbf{w}_{k-1} + y_k \boldsymbol{x}_k) = \mathbf{u}^T \mathbf{w}_{k-1} + y_k \mathbf{u}^T \mathbf{x}_k \geq \mathbf{u}^T \mathbf{w}_{k-1} + \gamma \geq k\gamma$$

Part 2:

$$\mathbf{w}_k^T \mathbf{w}_k = (\mathbf{w}_{k-1} + y_k \boldsymbol{x}_k)^T (\mathbf{w}_{k-1} + y_k \boldsymbol{x}_k)$$
$$= \mathbf{w}_{k-1}^T \mathbf{w}_{k-1} + 2y_k \mathbf{w}_{\boldsymbol{k-1}}^T \mathbf{x}_k + \mathbf{x}_k^T \mathbf{x}_k \leq \mathbf{w}_{k-1}^T \mathbf{w}_{k-1} + D^2 \leq kD^2$$

Putting it together: $\frac{\mathbf{u}^{\mathrm{T}}\mathbf{w}_k}{|\mathbf{u}||\mathbf{w}_k|} \geq \frac{k\gamma}{\sqrt{k}D}$, but this cannot exceed 1. so we must have $\frac{k\gamma}{\sqrt{k}D} \leq 1 \Rightarrow$

$$k \leq \left(\frac{D}{\gamma}\right)^2$$

$$k \leq \left(\frac{D}{\gamma}\right)^2$$

Margin

- γ is referred to as the **margin**
  - Min distance from data points to the decision boundary
  - Bigger margin -> easier classification problems -> faster convergence for perceptron

- This concept will be utilized later by support vector machines

Practical considerations for online perceptron

- The order of training examples matters!
  - Random is better
- When data is not linearly separable, no guarantee for convergence
- Rarely used in its vanilla form, which is primarily considered in theoretical analysis
- Simple modifications can significantly improve practical performance
  - Voted perceptron and average perceptron

Voted Perceptron

- Keep intermediate hypotheses and have them vote [Freund and Schapire 1998]

Let $\mathbf{w}_0 \leftarrow (0,0,0,\ldots,0)$

$c_0 = 0,\ t = 0$

Repeat for T times

  randomly shuffle training examples

  for each training example i :

    if $y_i \mathbf{w}_t^T \mathbf{x}_i \leq 0$

      $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + y_i \mathbf{x}_i$

      $t = t + 1$

      $c_t = 0$

    else $c_t = c_t + 1$

The output will be a collection of linear separators $\mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_M$ along with their survival time $c_0, c_1, \ldots, c_M$

The c's can be viewed as measures of the reliability of the $\mathbf{w}$'s

For classification, take a weighted vote among all separators:

$$\hat{y} = \text{sign}\{\sum_{t=0}^{M} c_t \text{sign}(\mathbf{w}_t^T \mathbf{x})\}$$

From Voted to Average Perceptron

- Voted perceptron requires storing all intermittent weights
  - Large memory consumption
  - Slow prediction time
  - The final boundary is no longer linear
- Average perceptron

$$\hat{\mathrm{y}} = \mathrm{sign}\{(\sum_{t=0}^{M} c_t \mathbf{w}_t^T)\mathbf{x}\}$$

  - Take the weighted average of all the intermittent weights
  - Can be implemented by maintaining an running average, no need to store all weights
  - Fast prediction time
  - Produce a linear decision boundary

Final Discussion on Perceptron

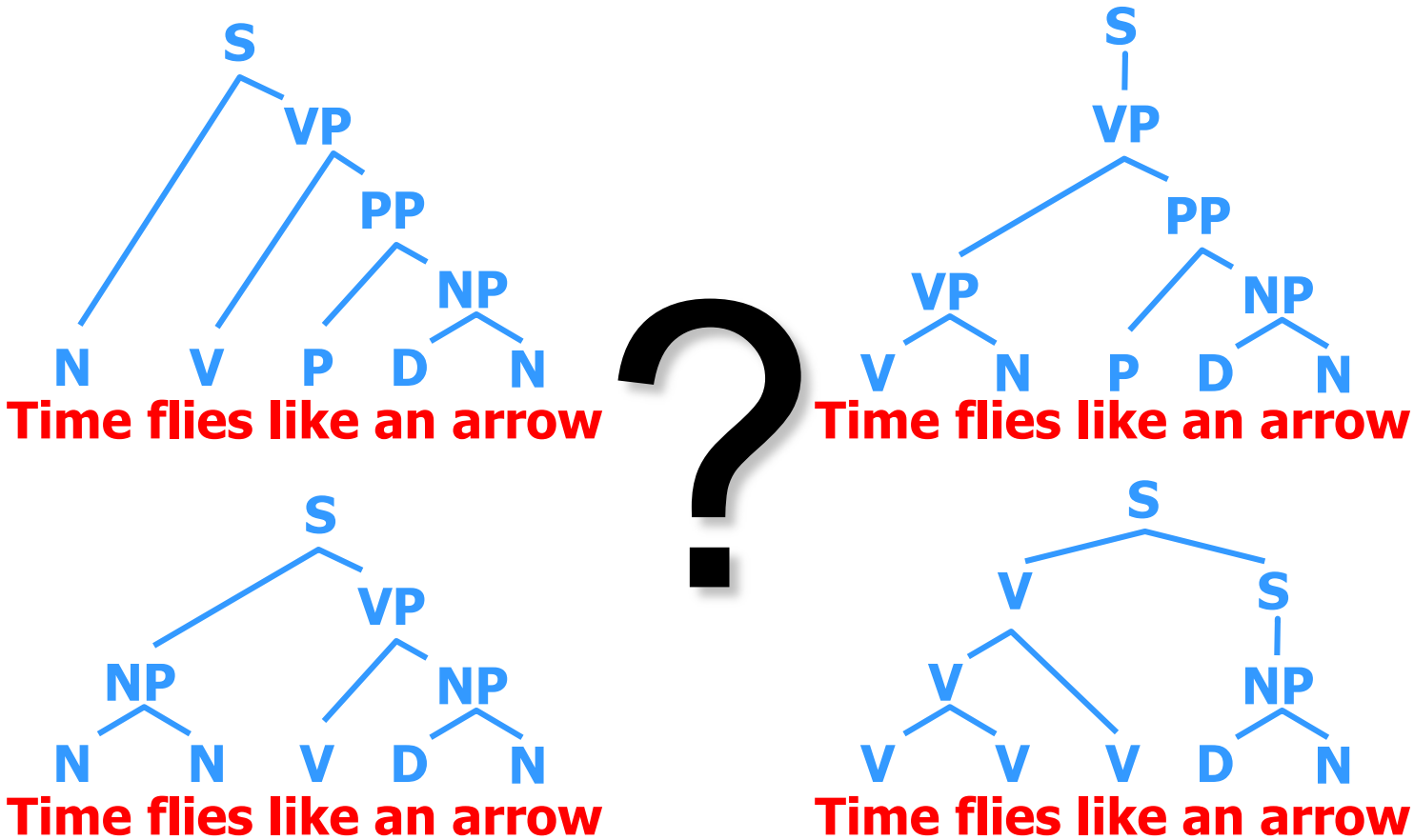
- Learns $\hat{y} = f(\mathbf{x})$ directly – a **discriminative** method

- Performs stochastic (sub-)gradient descent to optimize the perceptron loss (also called hinge loss with hinge=0)

- Guaranteed to converge in finite steps if the data is linearly separable
  - # of updates is inversely proportional to the **margin** of the optimal decision boundary $$k \leq \left(\frac{D}{\gamma}\right)^2$$
  - Guarantees convergence but not necessarily to the maximum margin separator – we will address this later in SVM
- Voted and average perceptrons provide significant performance improvement in practice

Beyond the Basic Perceptron

Structured Prediction with Perceptrons



...

Based on Jason Eisner's notes

A general problem

- Given some input x
  - An email, a sentence …
- Consider a set of candidate outputs y
  - Classifications for x (small number: often just 2)
  - POS Tags of x (exponentially many)
  - Parses of x (exponentially many)
  - Translations of x (exponentially many)
  - …

Structured prediction

- Want to find the "best" y, given x

Based on Jason Eisner's notes

Scoring by Linear Models

- Given some input x
- Consider a set of candidate outputs y
- Define a scoring function score(x,y)

Linear function: A sum of feature weights (you pick the features!)

Weight of feature k
(learned)

$$\text{score}(x, y) = \sum_{k} \theta_k f_k(x, y)$$

Ranges over all features,
e.g., k=5 (numbered features)
or k="see Det Noun" (named features)

Whether (x,y) has feature k(0 or 1)
Or **how many times** it fires (≥ 0)
Or **how strongly** it fires (real #)

- Choose y that maximizes score(x,y)

Based on Jason Eisner's notes

Scoring by Linear Models

- Given some input x
- Consider a set of candidate outputs y
- Define a scoring function score(x,y)

Linear function: A sum of feature weights (you pick the features!)

(learned)

$$\text{score}(x, y) = \boxed{\vec{\theta}} \cdot \vec{f}(x, y)$$

This linear decision rule is called a "perceptron."
It's a "structured perceptron" if it does structured prediction
(number of y candidates is unbounded, e.g., grows with |x|).

- Choose y that maximizes score(x,y)

Based on Jason Eisner's notes

Perceptron Training Algorithm

- initialize θ (usually to the zero vector)
- repeat:
  - Pick a training example (x,y)
  - Model predicts y^* that maximizes $\text{score}(x,y^*)$
  - Update weights by a step size $\varepsilon > 0$:

$$\theta = \theta + \varepsilon \cdot (f(x,y) - f(x,y^*))$$

If model prediction is correct ($y=y^*$), nothing happens

If model prediction was wrong ($y \neq y^*$), then we must have $\text{score}(x,y) \leq \text{score}(x,y^*)$ instead of $>$ as we want

Equivalently, $\theta \cdot f(x,y) \leq \theta \cdot f(x,y^*)$

Equivalently, $\theta \cdot (f(x,y) - f(x,y^*)) \leq 0$ but we want it positive.

Our update <u>increases</u> it (by $\varepsilon \cdot || f(x,y) - f(x,y^*) ||^2 \geq 0$)

Based on Jason Eisner's notes

Perceptron for Structured Prediction

- What we see here is the same as the regular perceptron
- Similar convergence guarantee
- The challenge is the inference part
  - Finding the y that maximizes the score for given x
  - Cannot resort to brute-force enumeration – dynamic programming is commonly used
  - Much research goes into
    - How to devise proper features and efficient algorithms for inference
    - How to perform approximate inference
    - How to learn when inference is approximate

Running Example: Predict a Tagging

Given word sequence x
Find max-score tag sequence y

score(**BOS** **N** **V** **EOS**) = ?
Time flies

$$\text{score}(x, y) = \sum_k \theta_k f_k(x, y)$$

So what are the features?
Let's start with the usual emission and transition features …

Running Example: Predict a Tagging

Given word sequence x

Find max-score tag sequence y

score(**BOS N V EOS**) = $\sum_k \theta_k f_k(x,y)$

Time flies

$= \theta_{BOS,N} + \theta_{N,Time} + \theta_{N,V} + \theta_{V,flies} + \theta_{V,EOS}$

So what are the features?

Let's start with the usual emission and transition features …

Running Example: Predict a Tagging

Given word sequence x

Find max-score tag sequence y

score(**BOS** **N** **V** **EOS**) = $\sum_k \theta_k f_k(x,y)$

Time flies

$= \theta_{BOS,N} + \theta_{N,Time} + \theta_{N,V} + \theta_{V,flies} + \theta_{V,EOS}$

For each $t \in$ Tags, $w \in$ Words:

define $f_{t,w}(x,y)$ = count of emission t w

$= |\{i: 1 \le i \le |x|, y_i = t, x_i = w\}|$

|Tags| |Words| emission features

For each t, t′ $\in$ Tags:

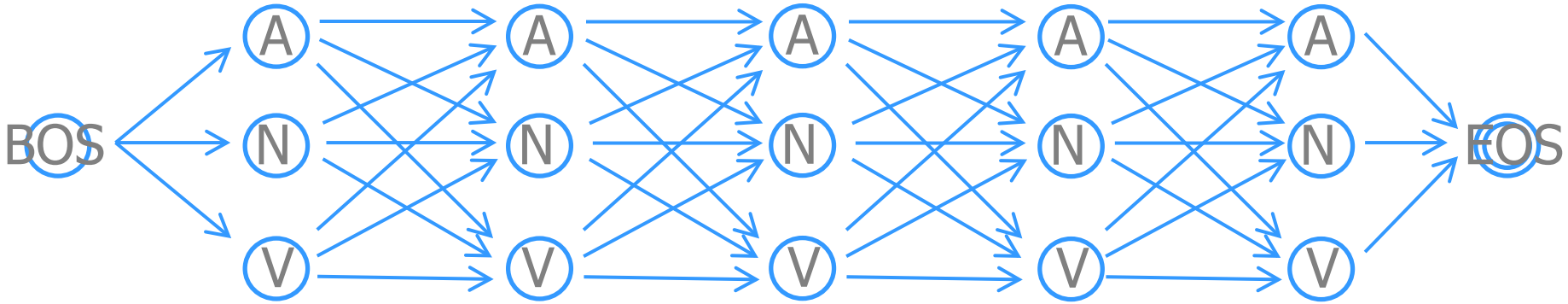
define $f_{t,t'}(x,y)$ = count of transition t t′

$= |\{i: 0 \le i \le |x|, y_i = t, y_{i+1} = t'\}|$

define $|Tags|^2$ transition features

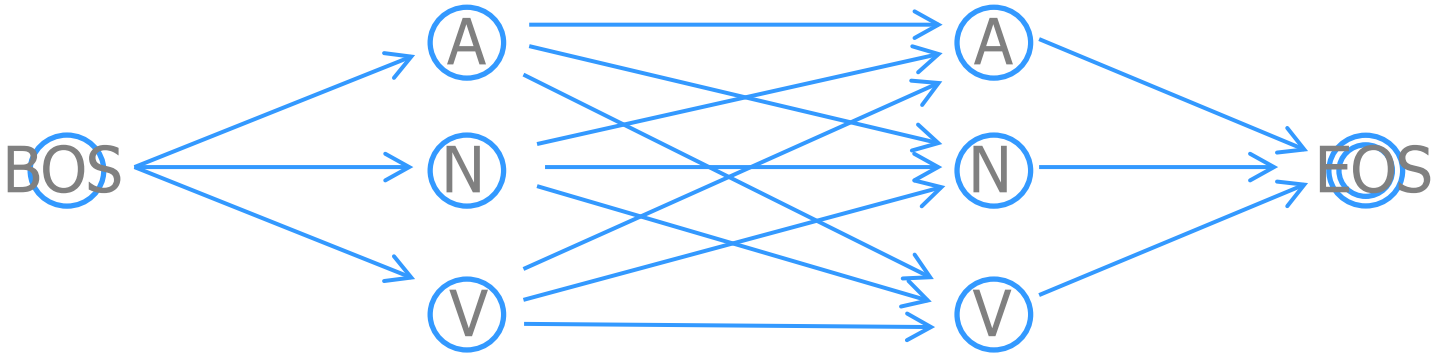
Running Example: Predict a Tagging

Lattice of exponentially many taggings (paths)
Viterbi algorithm can find the highest-scoring tagging



Running Example: Predict a Tagging

Lattice of exponentially many taggings (paths)
Viterbi algorithm can find the highest-scoring tagging



Running Example: Predict a Tagging

Lattice of exponentially many taggings (paths)
Viterbi algorithm can find the highest-scoring tagging
Set arc weights so that path weight = tagging score