Machine Learning 2

DS 4420 - Spring 2020

Neural Networks & backprop

Byron C Wallace

Neural Networks!

- In 2020, neural networks are the dominant technology in machine learning (for better or worse)!

Neural Networks!

- In 2020, neural networks are the dominant technology in machine learning (for better or worse)!

- Today, we'll go over some of the fundamentals of NNs and modern libraries (we saw a preview last week, with auto-diff)!

Neural Networks!

- In 2020, neural networks are the dominant technology in machine learning (for better or worse)!
- Today, we’ll go over some of the fundamentals of NNs and modern libraries (we saw a preview last week, with auto-diff)!
- This will also serve as a refresher on gradient descent

Gradient Descent in Linear Models

Last time we thought in **probabilistic terms** and discussed **maximum likelihood estimation** for "generative" models

Gradient Descent in Linear Models

Last time we thought in **probabilistic terms** and discussed **maximum likelihood estimation** for "generative" models

Today we'll take the view of learning as **search/ optimization**

Gradient Descent in Linear Models

Last time we thought in **probabilistic terms** and discussed **maximum likelihood estimation** for "generative" models

Today we'll take the view of learning as **search/optimization**

We'll start with linear models, review **gradient descent**, and then talk about **neural nets** + **backprop**

Loss

The simplest loss is probably 0/1 loss:

0 if we're correct
1 if we're wrong

What's an algo that minimizes this?

The ***Perceptron***!

Training data $\langle x, y \rangle$

Consider a simple linear model with parameters *w*

$$\hat{y}_i = \begin{cases} 1 & \text{if } \boxed{w \cdot x_i > 0} \\ -1 & \text{otherwise} \end{cases}$$

Training data $\langle x, y \rangle$

Consider a simple linear model with parameters *w*

$$\hat{y}_i = \begin{cases} 1 & \text{if } \boxed{w \cdot x_i > 0} \\ -1 & \text{otherwise} \end{cases}$$

(assumes bias term moved into *x* or omitted)

Training data $\langle x, y \rangle$

Consider a simple linear model with parameters *w*

$$\hat{y}_i = \begin{cases} 1 & \text{if } \boxed{w \cdot x_i > 0} \\ -1 & \text{otherwise} \end{cases}$$

(assumes bias term moved into *x* or omitted)

The learning problem is to estimate *w*

Training data $\langle x, y \rangle$

Consider a simple linear model with parameters *w*

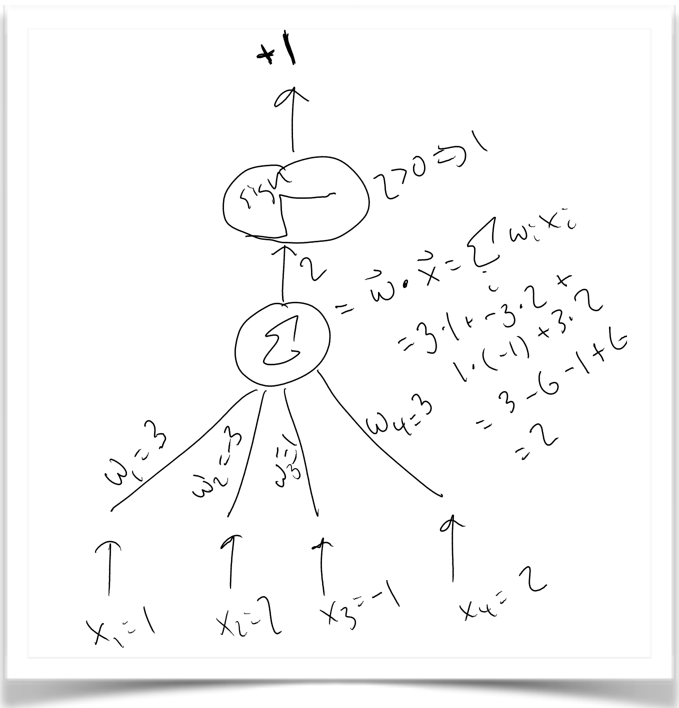
$$\hat{y}_i = \begin{cases} 1 & \text{if } \boxed{w \cdot x_i > 0} \\ -1 & \text{otherwise} \end{cases}$$

(assumes bias term moved into *x* or omitted)

The learning problem is to estimate *w*

What is our criterion for a good *w*? Minimal ***loss***

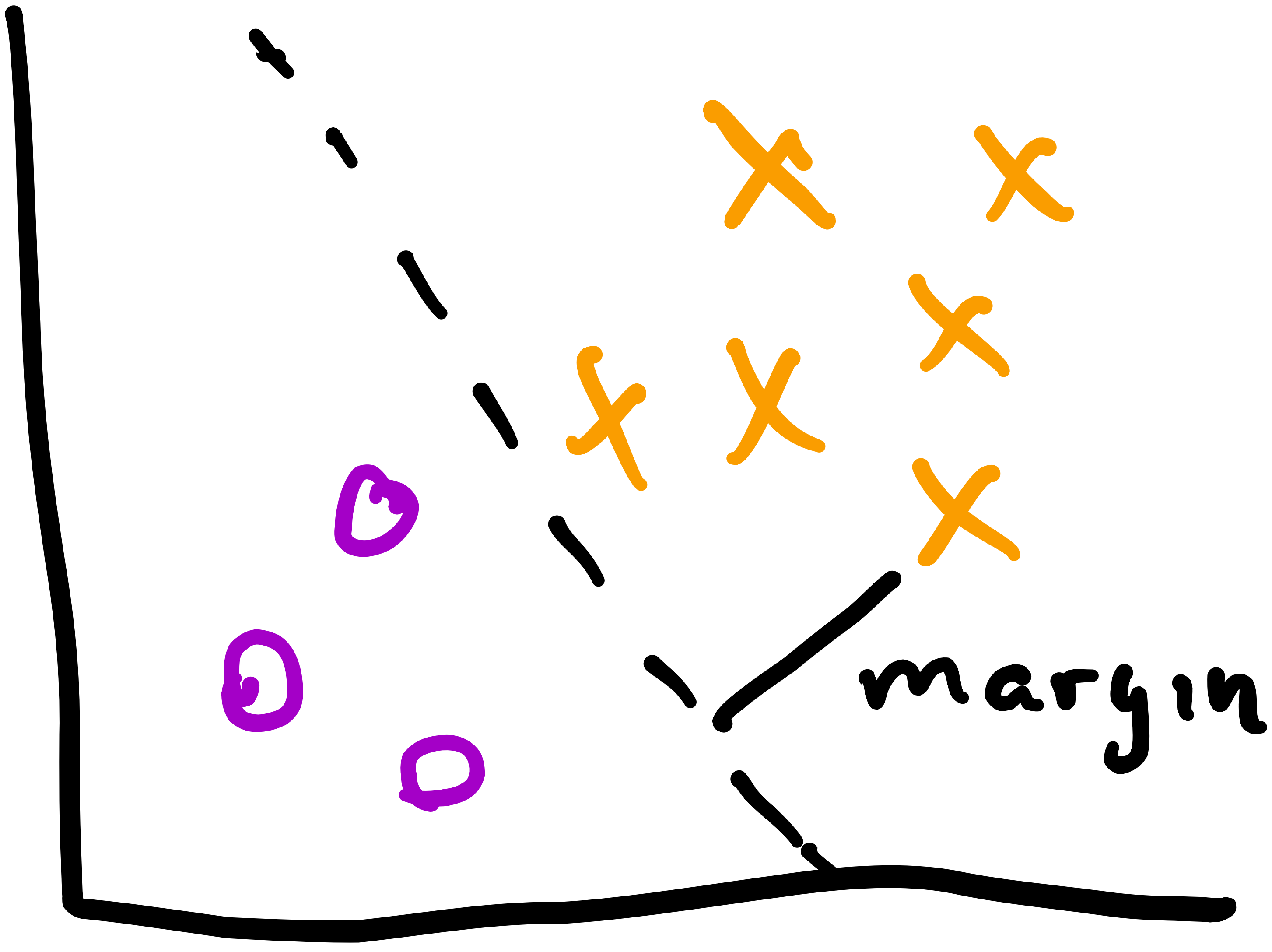
Perceptron!



Algorithm 5 PerceptronTrain($\mathbf{D}$, $MaxIter$)

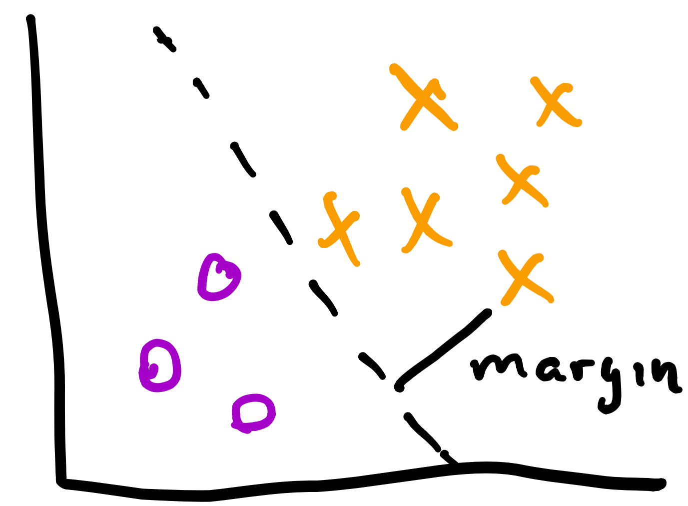
$w_d \leftarrow 0$, for all $d = 1 \dots D$ // initialize weights
$b \leftarrow 0$ // initialize bias
for $iter = 1 \dots MaxIter$ **do**
  for all $(\boldsymbol{x}, y) \in \mathbf{D}$ **do**
    $a \leftarrow \sum_{d=1}^{D} w_d\, x_d + b$ // compute activation for this example
    if $ya \leq 0$ **then**
      $w_d \leftarrow w_d + y x_d$, for all $d = 1 \dots D$ // update weights
      $b \leftarrow b + y$ // update bias
    end if
  end for
end for
return $w_0, w_1, \dots, w_D, b$

Fig and Alg from CIML [Daume]


margin

Problems with 0/1 loss

- If we're wrong by .0001 it is "as bad" as being wrong by .9999
- Because it is discrete, optimization is hard if the instances are not linearly separable



Smooth loss

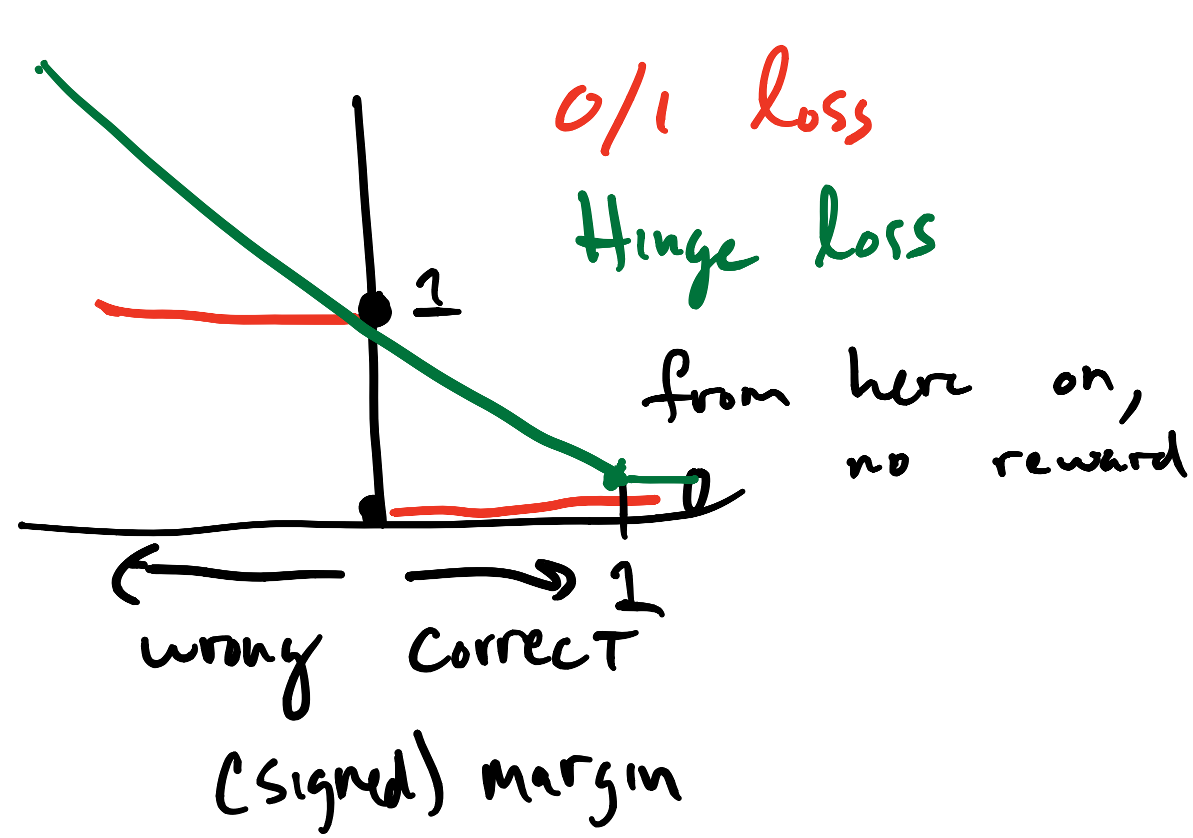
Idea: Introduce a “smooth” loss function to make optimization easier

Example: Hinge loss

$$\mathcal{L}_{\text{Hinge}}(y,z) = \max\{0, 1 - y\cdot z\}$$

$y \in \{1,-1\}$

$z = w\cdot x_i$ ("raw" output)



Losses

Zero/one:	$\ell^{(0/1)}(y, \hat{y}) = \mathbf{1}[y\hat{y} \leq 0]$
Hinge:	$\ell^{(\text{hin})}(y, \hat{y}) = \max\{0, 1 - y\hat{y}\}$
Logistic:	$\ell^{(\log)}(y, \hat{y}) = \frac{1}{\log 2} \log\left(1 + \exp[-y\hat{y}]\right)$
Exponential:	$\ell^{(\exp)}(y, \hat{y}) = \exp[-y\hat{y}]$
Squared:	$\ell^{(\text{sqr})}(y, \hat{y}) = (y - \hat{y})^2$

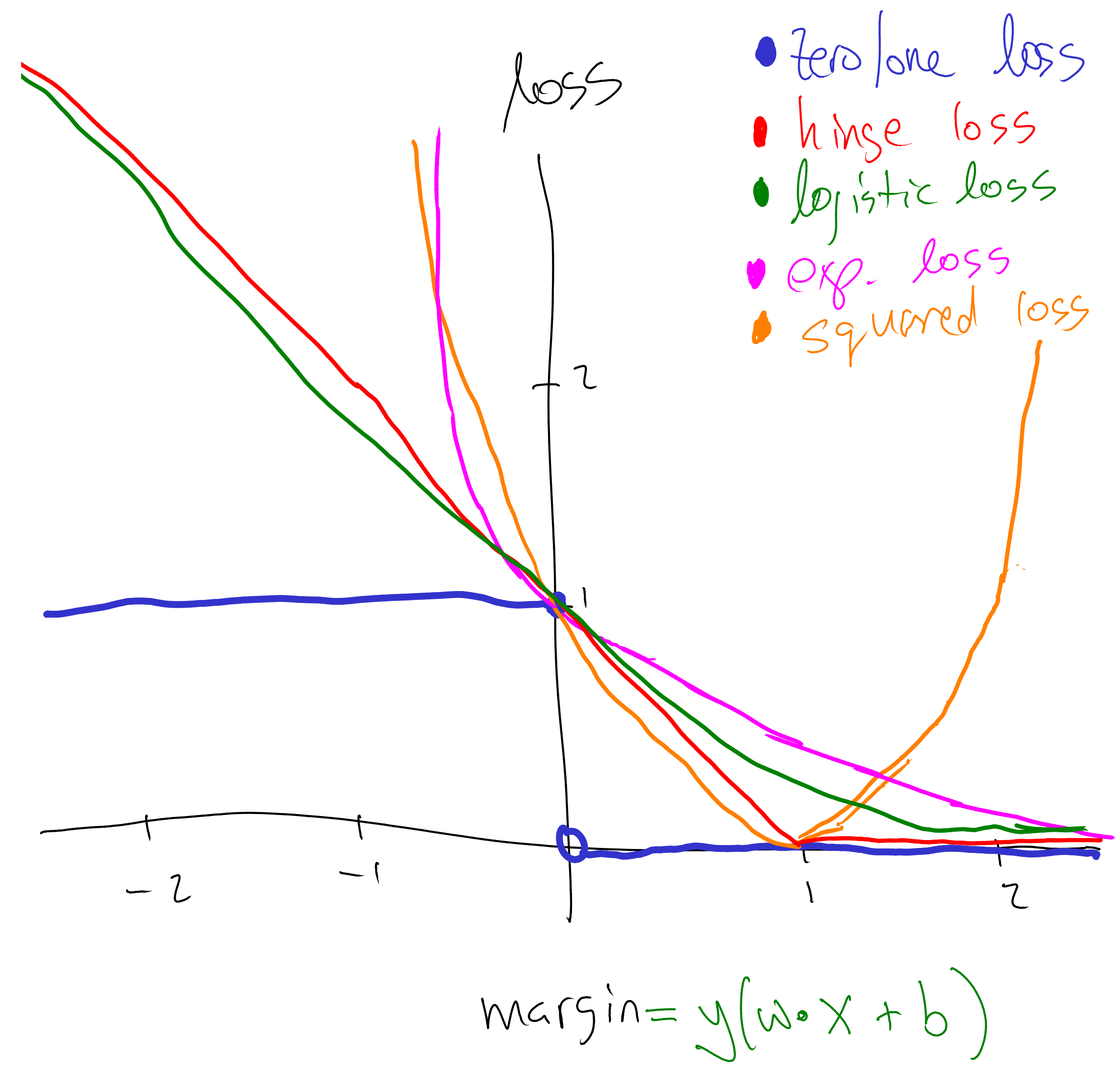


Fig and Eq's from CIML [Daume]

Regularization

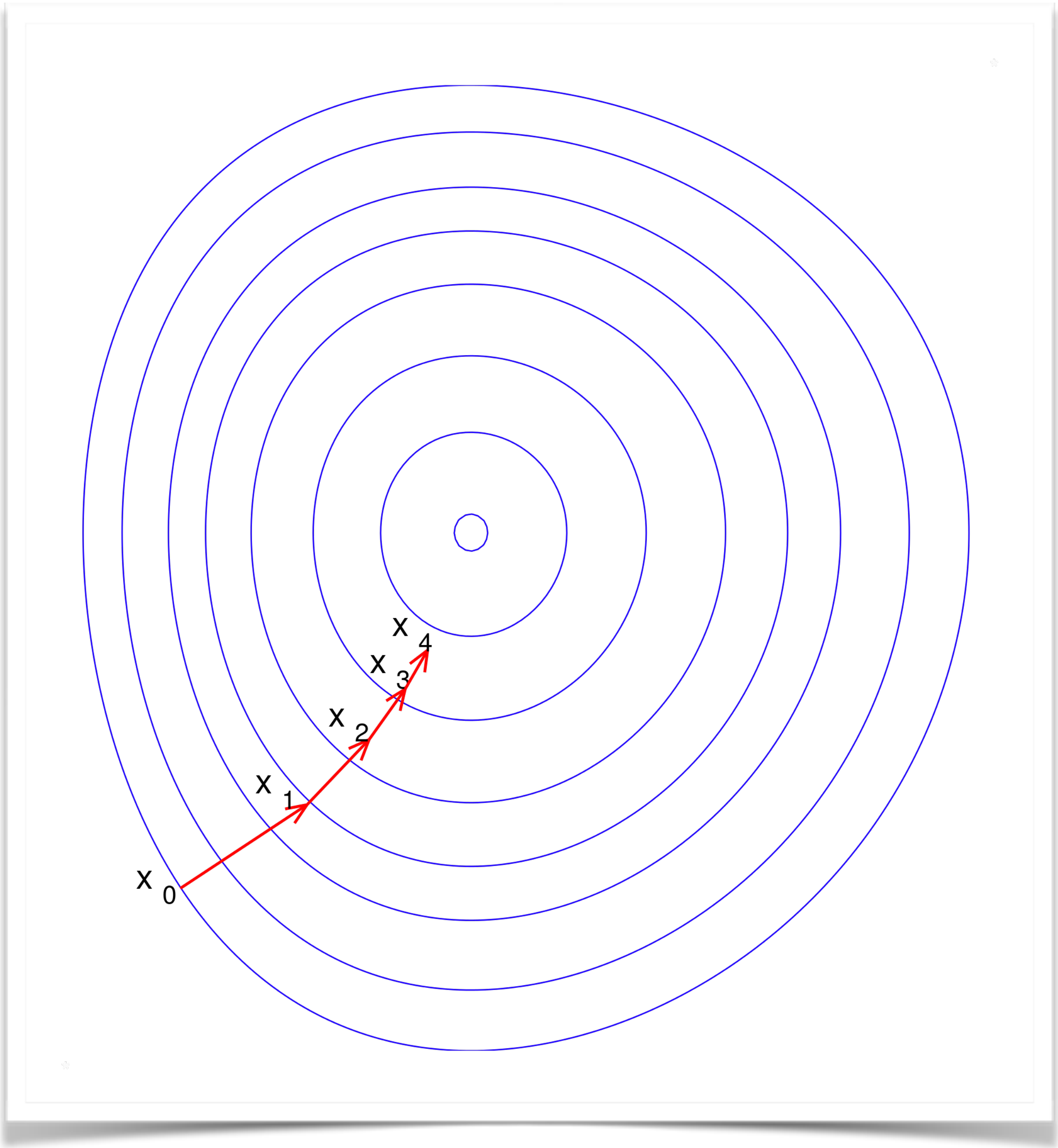
$$\min_{\boldsymbol{w},b} \quad \sum_n \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n + b)$$

Regularization

$$\min_{\boldsymbol{w},b} \quad \sum_n \ell(y_n, \boldsymbol{w}\cdot\boldsymbol{x}_n + b) + \boxed{\lambda R(\boldsymbol{w}, b)}$$

Prevent *w* from “getting to crazy”

Gradient descent



Algorithm 21 GradientDescent($\mathcal{F}$, K, η_1, ...)

```
z^(0) ← ⟨0, 0, ..., 0⟩                          // initialize variable we are optimizing
for k = 1 ... K do
    g^(k) ← ∇_z F |_z^(k-1)                     // compute gradient at current location
    z^(k) ← z^(k-1) − η^(k) g^(k)               // take a step down the gradient
end for
return z^(K)
```

Alg from CIML [Daume]

$$\nabla_{\boldsymbol{w}} \mathcal{L} = \nabla_{\boldsymbol{w}} \sum_n \exp\left[- y_n (\boldsymbol{w} \cdot \boldsymbol{x}_n + b) \right] + \nabla_{\boldsymbol{w}} \frac{\lambda}{2} \, ||\boldsymbol{w}||^2$$

$$
\begin{aligned}
\nabla_{\boldsymbol{w}} \mathcal{L} &= \nabla_{\boldsymbol{w}} \sum_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \nabla_{\boldsymbol{w}} \frac{\lambda}{2} \left|\left|\boldsymbol{w}\right|\right|^2 \\
&= \sum_n \left(\nabla_{\boldsymbol{w}} - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right) \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \lambda \boldsymbol{w}
\end{aligned}
$$

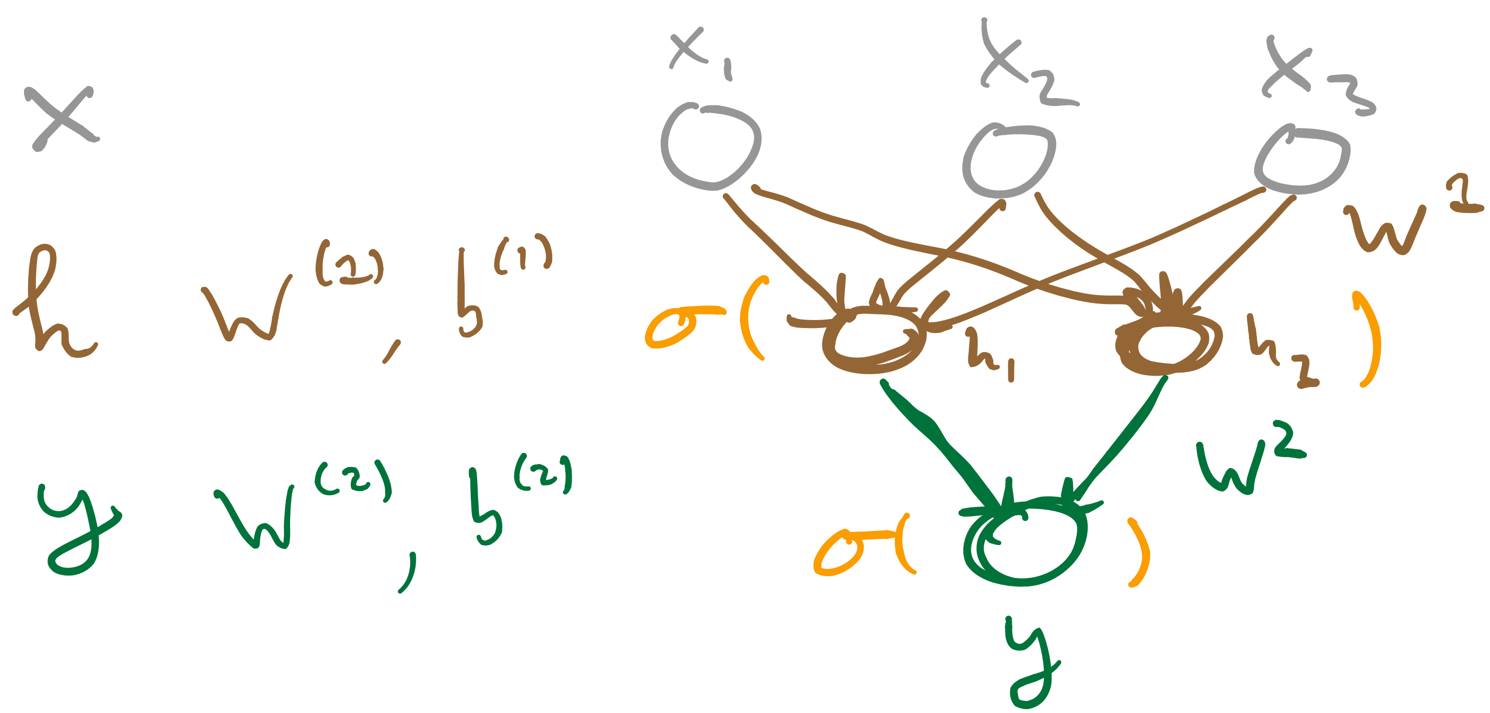
$$
\begin{aligned}
\nabla_{\boldsymbol{w}}\mathcal{L} &= \nabla_{\boldsymbol{w}} \sum_n \exp\big[-y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b)\big] + \nabla_{\boldsymbol{w}} \frac{\lambda}{2}\,||\boldsymbol{w}||^2 \\
&= \sum_n \left(\nabla_{\boldsymbol{w}} - y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b)\right) \exp\big[-y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b)\big] + \lambda\boldsymbol{w} \\
&= -\sum_n y_n \boldsymbol{x}_n \exp\big[-y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b)\big] + \lambda\boldsymbol{w}
\end{aligned}
$$

Limitations of linear models

Neural networks

Idea: Basically stack together a bunch of linear models.

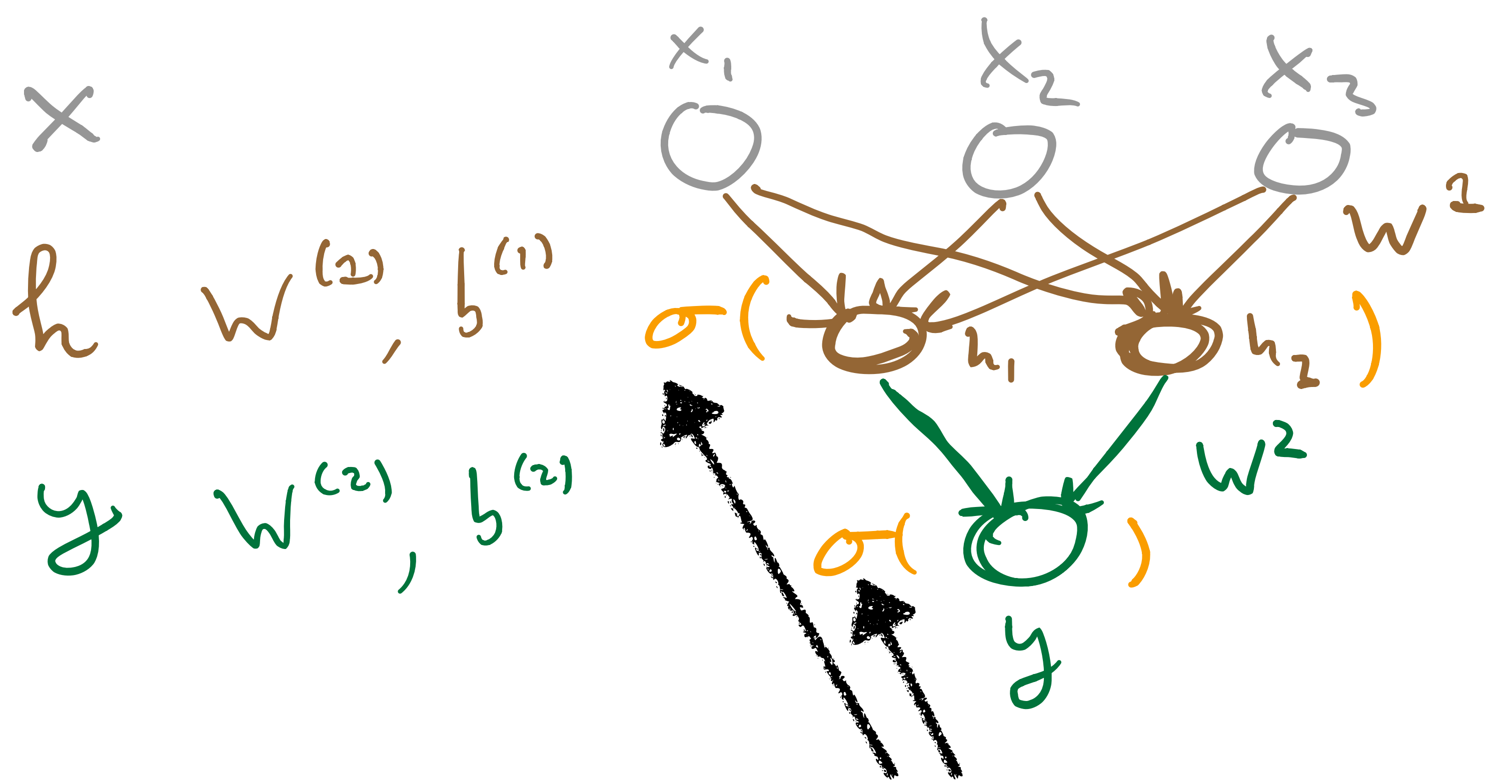
This introduces *hidden units* which are neither observations (x) nor outputs (y)



Neural networks

Idea: Basically stack together a bunch of linear models.

This introduces *hidden units* which are neither observations (x) nor outputs (y)

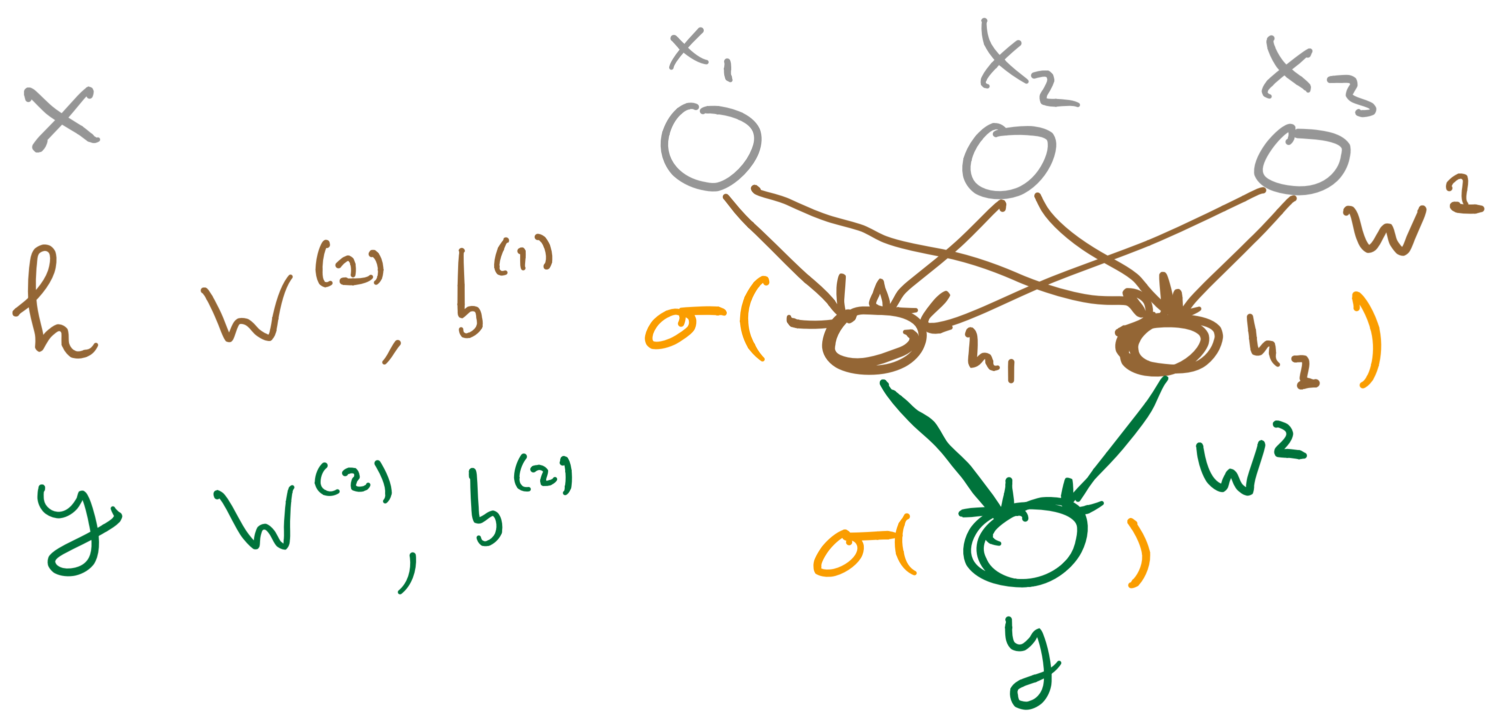


(Non-linear) activation functions

Neural networks

Idea: Basically stack together a bunch of linear models.

This introduces *hidden units* which are neither observations (x) nor outputs (y)

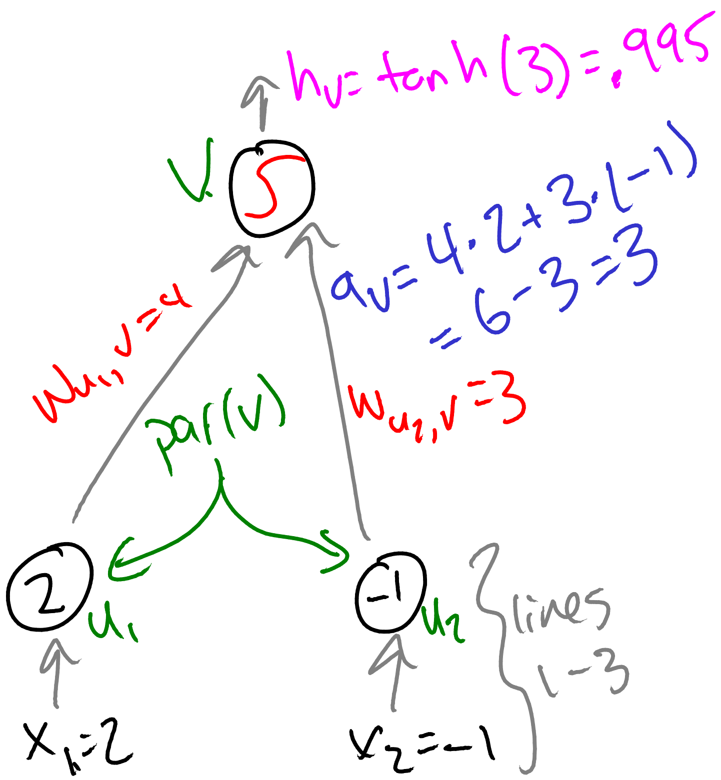


The challenge: How do we update weights associated with each node in this *multi-layer* regime?

back-propagation $=$ gradient descent $+$ chain rule

Algorithm 27 FORWARDPROPAGATION($\boldsymbol{x}$)

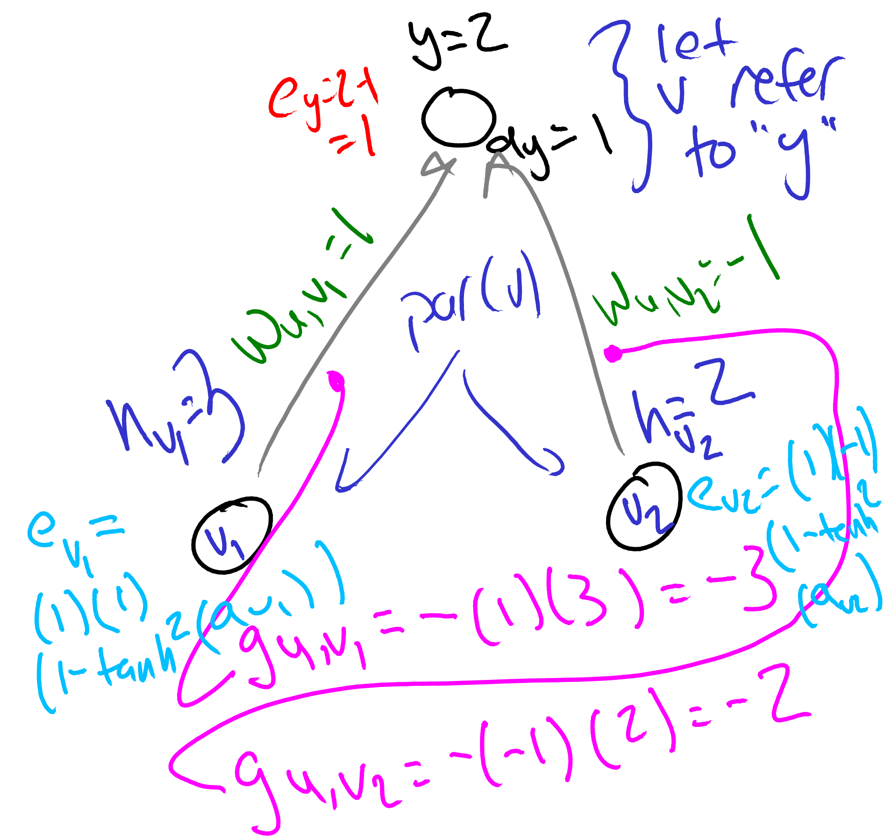
```
1: for all input nodes u do
2:     h_u ← corresponding feature of x
3: end for
4: for all nodes v in the network whose parent's are computed do
5:     a_v ← Σ_{u∈par(v)} w_(u,v) h_u
6:     h_v ← tanh(a_v)
7: end for
8: return a_y
```



Tanh is another common activation function

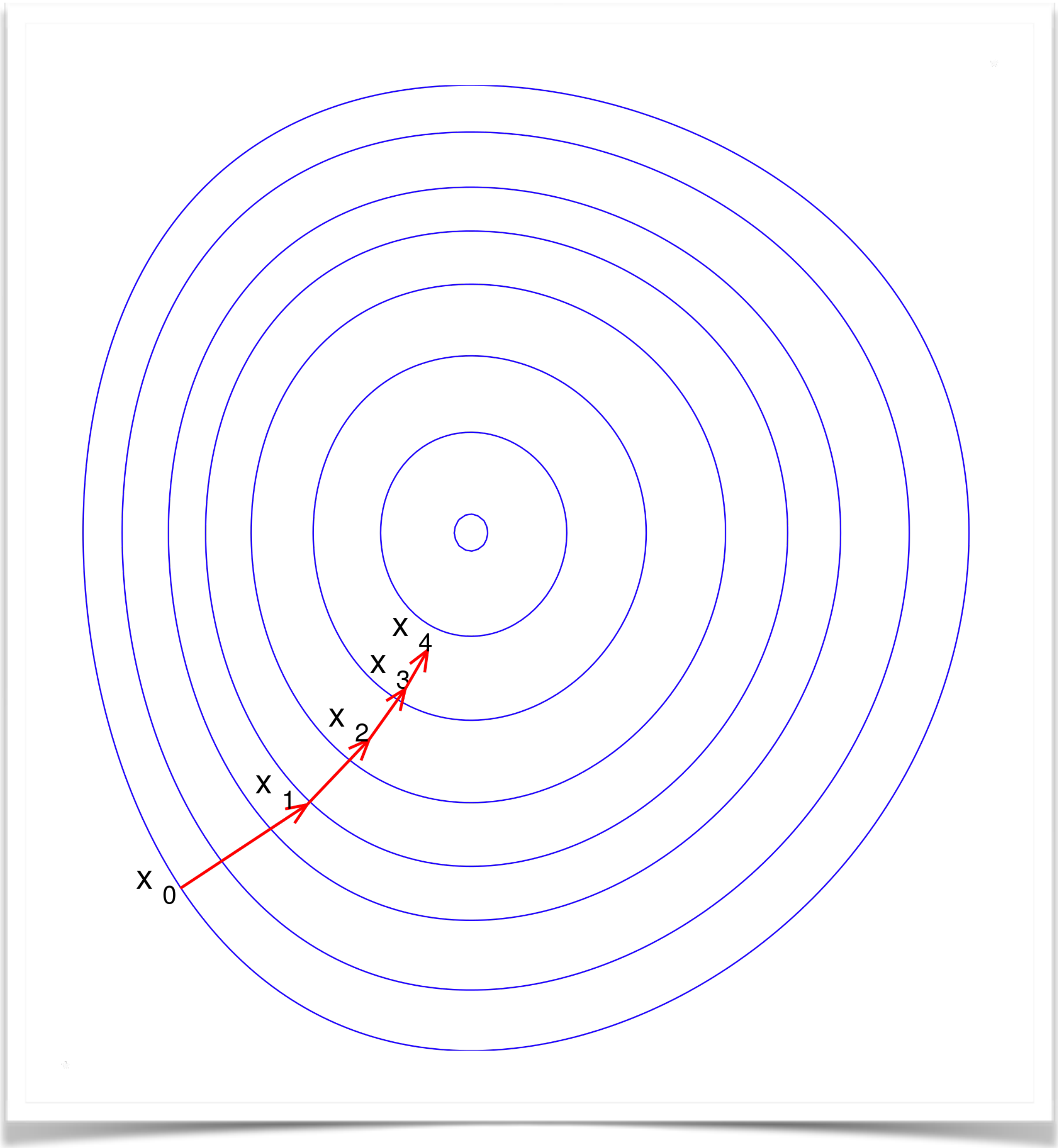
Algorithm 28 BackPropagation(x, y)

run ForwardPropagation(x) to compute activations
$e_y \leftarrow y - a_y$ // compute overall network error
for all nodes v in the network whose error e_v is computed **do**
  for all $u \in par(v)$ **do**
    $g_{u,v} \leftarrow -e_v h_u$ // compute gradient of this edge
    $e_u \leftarrow e_u + e_v w_{u,v}(1 - \tanh^2(a_u))$ // compute the "error" of the parent node
  end for
end for
return all gradients g_e



What are we doing with these gradients again?

Gradient descent



By Gradient_descent.png: The original uploader was Olegalexandrov at English Wikipedia.derivative work: Zerodamage - This file was derived from: Gradient descent.png:, Public Domain, https://commons.wikimedia.org/w/index.php?curid=20569355

Neural Networks!

If you're interested in learning more…

DS4440 // practical neural networks // spring 2019