CSCI567 Machine Learning (Spring 2025)

Haipeng Luo

University of Southern California

Feb 07, 2025

Administration

HW1 was due yesterday. Remember: only one late day allowed

HW2 will be released next week.

Outline

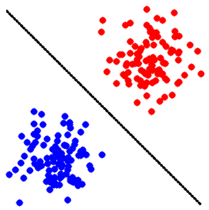
Outline

Linear classifiers

Linear models for **binary** classification:

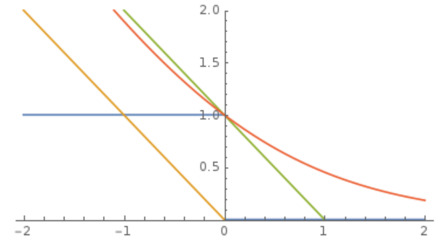
Step 1. Model is the set of **separating hyperplanes**

$$\mathcal{F} = \{f(\boldsymbol{x}) = \mathsf{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}) \mid \boldsymbol{w} \in \mathbb{R}^{\mathsf{D}}\}$$



Linear classifiers

Step 2. Pick the **surrogate loss**



- perceptron loss $\ell_{\text{perceptron}}(z) = \max\{0, -z\}$ (used in Perceptron)
- hinge loss $\ell_{\text{hinge}}(z) = \max\{0, 1 - z\}$(used in SVM and many others)
- logistic loss $\ell_{\text{logistic}}(z) = \log(1 + \exp(-z))$ (used in logistic regression)

Linear classifiers

Step 3. Find empirical risk minimizer (ERM):

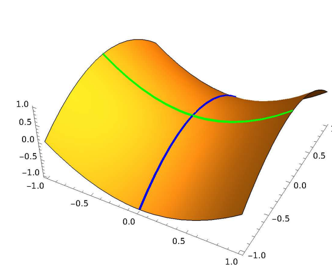
$$\boldsymbol{w}^* = \operatorname*{argmin}_{\boldsymbol{w} \in \mathbb{R}^{\mathsf{D}}} F(\boldsymbol{w}) = \operatorname*{argmin}_{\boldsymbol{w} \in \mathbb{R}^{\mathsf{D}}} \frac{1}{N} \sum_{n=1}^{N} \ell(y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n)$$

using

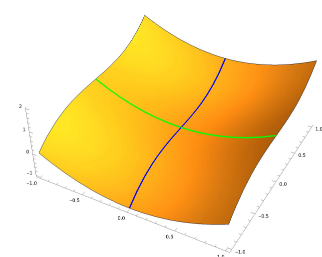
- **GD:** $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla F(\boldsymbol{w})$
- **SGD:** $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \tilde{\nabla} F(\boldsymbol{w})$ $\qquad (\mathbb{E}[\tilde{\nabla} F(\boldsymbol{w})] = \nabla F(\boldsymbol{w}))$
- **Newton:** $\boldsymbol{w} \leftarrow \boldsymbol{w} - \left(\nabla^2 F(\boldsymbol{w})\right)^{-1} \nabla F(\boldsymbol{w})$

Convergence guarantees of GD/SGD

- GD/SGD converges to a stationary point
- for convex objectives, this is all we need
- for nonconvex objectives, can get stuck at local minimizers or "bad" saddle points (random initialization escapes "good" saddle points)



"good" saddle points



"bad" saddle points

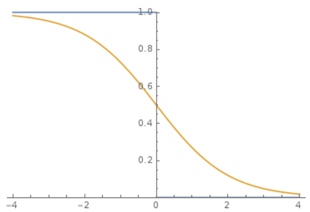
Perceptron and logistic regression

Initialize $\boldsymbol{w} = \boldsymbol{0}$ or randomly.

Repeat:

- pick a data point $\boldsymbol{x}_n$ uniformly at random (**common trick for SGD**)
- update parameter:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \begin{cases} \mathbb{I}[y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n \leq 0] y_n \boldsymbol{x}_n & \text{(Perceptron)} \\ \eta \sigma(-y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n) y_n \boldsymbol{x}_n & \text{(logistic regression)} \end{cases}$$



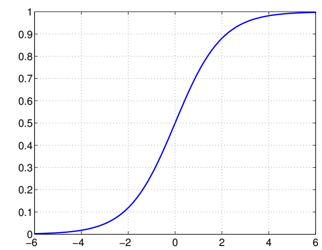
A Probabilistic view of logistic regression

Minimizing logistic loss = MLE for the sigmoid model

$$\boldsymbol{w}^* = \operatorname*{argmin}_{\boldsymbol{w}} \sum_{n=1}^{N} \ell_{\text{logistic}}(y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n) = \operatorname*{argmax}_{\boldsymbol{w}} \prod_{n=1}^{N} \mathbb{P}(y_n \mid \boldsymbol{x_n}; \boldsymbol{w})$$

where

$$\mathbb{P}(y \mid \boldsymbol{x}; \boldsymbol{w}) = \sigma(y \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}) = \frac{1}{1 + e^{-y \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}}}$$



Outline

Classification

Recall the setup:

- input (feature vector): $\boldsymbol{x} \in \mathbb{R}^{\mathsf{D}}$
- output (label): $y \in [\mathsf{C}] = \{1, 2, \cdots, \mathsf{C}\}$
- goal: learn a mapping $f : \mathbb{R}^{\mathsf{D}} \rightarrow [\mathsf{C}]$

Examples:

- recognizing digits ($\mathsf{C} = 10$) or letters ($\mathsf{C} = 26$ or 52)
- predicting weather: sunny, cloudy, rainy, etc
- predicting image category: ImageNet dataset ($\mathsf{C} \approx 20K$)

Nearest Neighbor Classifier naturally works for arbitrary C.

Linear models: from binary to multiclass

Step 1: *What should a linear model look like for multiclass tasks?*

Note: a linear model for binary tasks (switching from $\{-1,+1\}$ to $\{1,2\}$)

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \geq 0 \\ 2 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} < 0 \end{cases}$$

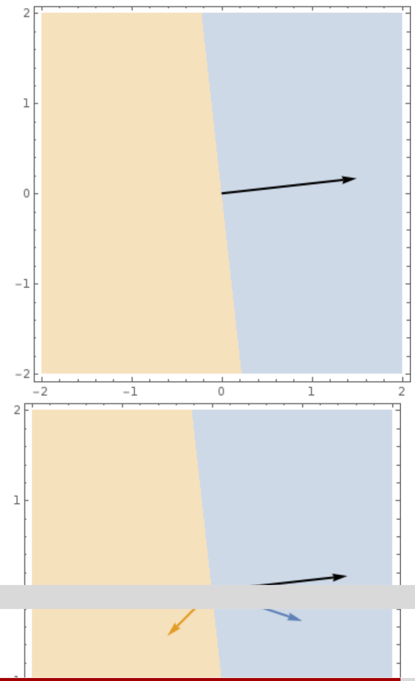
can be written as

$$\begin{aligned} f(\boldsymbol{x}) &= \begin{cases} 1 & \text{if } \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \geq \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} \\ 2 & \text{if } \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} > \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \end{cases} \\ &= \operatorname*{argmax}_{k \in \{1,2\}} \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x} \end{aligned}$$

for any $\boldsymbol{w}_1, \boldsymbol{w}_2$ s.t. $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$

Think of $\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}$ as **a score for class k**.

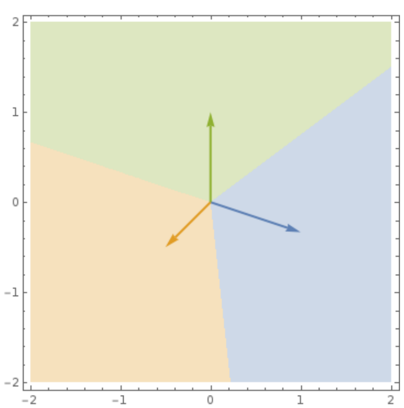
Linear models: from binary to multiclass



$\boldsymbol{w} = (\frac{3}{2}, \frac{1}{6}) = \boldsymbol{w}_1 - \boldsymbol{w}_2$
$\boldsymbol{w}_1 = (1, -\frac{1}{3})$
$\boldsymbol{w}_2 = (-\frac{1}{2}, -\frac{1}{2})$

- Blue class:
  $\{\boldsymbol{x} : 1 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$

Linear models: from binary to multiclass



$\boldsymbol{w}_1 = (1, -\frac{1}{3})$
$\boldsymbol{w}_2 = (-\frac{1}{2}, -\frac{1}{2})$
$\boldsymbol{w}_3 = (0, 1)$

- Blue class:
  $\{\boldsymbol{x} : 1 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$
- Orange class:
  $\{\boldsymbol{x} : 2 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$
- Green class:
  $\{\boldsymbol{x} : 3 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$

Linear models for multiclass classification

$$\begin{aligned} \mathcal{F} &= \left\{ f(\boldsymbol{x}) = \operatorname*{argmax}_{k \in [\mathsf{C}]} \ \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x} \mid \boldsymbol{w}_1, \ldots, \boldsymbol{w}_{\mathsf{C}} \in \mathbb{R}^{\mathsf{D}} \right\} \\ &= \left\{ f(\boldsymbol{x}) = \operatorname*{argmax}_{k \in [\mathsf{C}]} \ (\boldsymbol{W}\boldsymbol{x})_k \mid \boldsymbol{W} \in \mathbb{R}^{\mathsf{C} \times \mathsf{D}} \right\} \end{aligned}$$

Step 2: *How do we generalize perceptron/hinge/logistic loss?*

This lecture: focus on the more popular **logistic loss**

Multinomial logistic regression: a probabilistic view

Observe: for binary logistic regression, with $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$:

$$\mathbb{P}(y = 1 \mid \boldsymbol{x}; \boldsymbol{w}) = \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}}} = \frac{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}}}{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}} + e^{\boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x}}} \propto e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}}$$

Naturally, for multiclass:

$$\mathbb{P}(y = k \mid \boldsymbol{x}; \boldsymbol{W}) = \frac{e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}}{\sum_{k' \in [\mathsf{C}]} e^{\boldsymbol{w}_{k'}^{\mathrm{T}}\boldsymbol{x}}} \propto e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}$$

This is called the *softmax function*.

Applying MLE again

Maximize probability of seeing labels $y_1, \ldots, y_{\mathsf{N}}$ given $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{\mathsf{N}}$

$$P(\boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \mathbb{P}(y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \frac{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}}\boldsymbol{x}_n}}{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}_n}}$$

By taking **negative log**, this is equivalent to minimizing

$$F(\boldsymbol{W}) = \sum_{n=1}^{\mathsf{N}} \ln\left(\frac{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}_n}}{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}}\boldsymbol{x}_n}}\right) = \sum_{n=1}^{\mathsf{N}} \ln\left(1 + \sum_{k \neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}\right)$$

This is the *multiclass logistic loss*, a.k.a. *cross-entropy loss*.

When $\mathsf{C} = 2$, this is the same as binary logistic loss.

Step 3: Optimization

Apply **SGD**: what is the gradient of

$$F_n(\boldsymbol{W}) = \ln\left(1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}\right)?$$

It's a $\mathsf{C} \times \mathsf{D}$ matrix. Let's focus on the k-th row:

If $k \neq y_n$:

$$\nabla_{\boldsymbol{w}_k^{\mathrm{T}}} F_n(\boldsymbol{W}) = \frac{e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}}{1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}} = \mathbb{P}(k \mid \boldsymbol{x}_n; \boldsymbol{W})\boldsymbol{x}_n^{\mathrm{T}}$$

else:

$$\nabla_{\boldsymbol{w}_k^{\mathrm{T}}} F_n(\boldsymbol{W}) = \frac{-\left(\sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}\right)}{1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}} = \left(\mathbb{P}(y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) - 1\right)\boldsymbol{x}_n^{\mathrm{T}}$$

SGD for multinomial logistic regression

Initialize $\boldsymbol{W} = \boldsymbol{0}$ (or randomly). Repeat:

1. pick $n \in [\mathsf{N}]$ uniformly at random
2. update the parameters

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \begin{pmatrix} \mathbb{P}(y = 1 \mid \boldsymbol{x}_n; \boldsymbol{W}) \\ \vdots \\ \mathbb{P}(y = y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) - 1 \\ \vdots \\ \mathbb{P}(y = \mathsf{C} \mid \boldsymbol{x}_n; \boldsymbol{W}) \end{pmatrix} \boldsymbol{x}_n^{\mathrm{T}}$$

Think about why the algorithm makes sense intuitively.

A note on prediction

Having learned $\boldsymbol{W}$, we can either

- make a *deterministic* prediction $\operatorname{argmax}_{k\in[\mathsf{C}]}\ \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}$
- make a *randomized* prediction according to $\mathbb{P}(k \mid \boldsymbol{x}; \boldsymbol{W}) \propto e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}$

Generalization of cross-entropy loss

Given a general model class:

$$\mathcal{F} = \left\{ f(\boldsymbol{x}) = \underset{k\in[\mathsf{C}]}{\operatorname{argmax}}\ s_k(\boldsymbol{x}) \right\}$$

where s_k is the **"scoring" function** for class k.

The cross-entropy loss of f for a training sample $(\boldsymbol{x}, y)$ is

$$-\ln\left(\frac{e^{s_y(\boldsymbol{x})}}{\sum_{k\in[\mathsf{C}]} e^{s_k(\boldsymbol{x})}}\right) = \ln\left(1 + \sum_{k\neq y} e^{s_k(\boldsymbol{x}) - s_y(\boldsymbol{x})}\right)$$

Reduce multiclass to binary

Is there an *even more general and simpler approach* to derive multiclass classification algorithms?

Given a binary classification algorithm (*any one*, not just linear methods), can we turn it to a multiclass algorithm, *in a black-box manner*?

Yes, there are in fact many ways to do it.

- **one-versus-all** (one-versus-rest, one-against-all, etc.)
- **one-versus-one** (all-versus-all, etc.)
- **Error-Correcting Output Codes** (ECOC)
- **tree-based reduction**

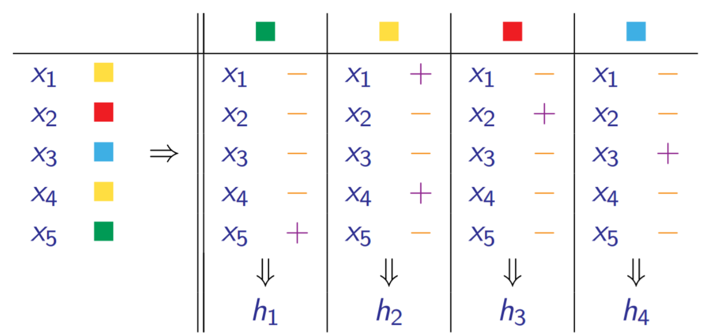
One-versus-all (OvA)

(picture credit: link)

Idea: train C binary classifiers to learn "**is class k or not?**" for each k.

Training: for each class $k \in [\mathsf{C}]$,

- relabel examples with class k as $+1$, and all others as -1
- train a binary classifier h_k using this new dataset



One-versus-all (OvA)

Prediction: for a new example $\boldsymbol{x}$

- ask each h_k: **does this belong to class k?** (i.e. $h_k(\boldsymbol{x})$)
- randomly pick among all k's s.t. $h_k(\boldsymbol{x}) = +1$.

Issue: will (probably) make a mistake *as long as one of h_k errs*.

One-versus-one (OvO)

(picture credit: link)

Idea: train $\binom{C}{2}$ binary classifiers to learn "**is class k or k'?**".

Training: for each pair (k, k'),

- relabel examples with class k as $+1$ and examples with class k' as -1
- *discard all other examples*
- train a binary classifier $h_{(k,k')}$ using this new dataset

			🟩 vs. 🟨	🟩 vs. 🟥	🟥 vs. 🟦	🟦 vs. 🟨	🟩 vs. 🟦	🟥 vs. 🟨
x_1	🟨		x_1 −			x_1 −		x_1 −
x_2	🟥			x_2 −	x_2 +			x_2 +
x_3	🟦	$\Rightarrow$			x_3 −	x_3 +	x_3 −	
x_4	🟨		x_4 −			x_4 −		x_4 −
x_5	🟩		x_5 +	x_5 +			x_5 +	
			$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
			$h_{(1,2)}$	$h_{(1,3)}$	$h_{(3,4)}$	$h_{(4,2)}$	$h_{(1,4)}$	$h_{(3,2)}$

One-versus-one (OvO)

Prediction: for a new example $\boldsymbol{x}$

- ask each classifier $h_{(k,k')}$ to **vote for either class k or k'**
- predict the class with the most votes (break tie in some way)

More robust than one-versus-all, but *slower* in prediction.

Error-correcting output codes (ECOC)

(picture credit: link)

Idea: based on a code $\boldsymbol{M} \in \{-1, +1\}^{\mathsf{C}\times\mathsf{L}}$, train L binary classifiers to learn "**is bit b on or off**".

Training: for each bit $b \in [\mathsf{L}]$

- relabel example x_n as $M_{y_n,b}$
- train a binary classifier h_b using this new dataset.

M	1	2	3	4	5
🟩	+	−	+	−	+
🟨	−	−	+	+	+
🟥	+	+	−	−	−
🟦	+	+	+	+	−

			1	2	3	4	5
x_1	🟨		x_1 −	x_1 −	x_1 +	x_1 +	x_1 +
x_2	🟥		x_2 +	x_2 +	x_2 −	x_2 −	x_2 −
x_3	🟦	$\Rightarrow$	x_3 +	x_3 +	x_3 +	x_3 +	x_3 −
x_4	🟨		x_4 −	x_4 −	x_4 +	x_4 +	x_4 +
x_5	🟩		x_5 +	x_5 −	x_5 +	x_5 −	x_5 +
			$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
			h_1	h_2	h_3	h_4	h_5

Error-correcting output codes (ECOC)

Prediction: for a new example $\boldsymbol{x}$

- compute the **predicted code** $\boldsymbol{c} = (h_1(\boldsymbol{x}), \ldots, h_{\mathsf{L}}(\boldsymbol{x}))^{\mathrm{T}}$
- predict the class with the **most similar code**: $k = \mathrm{argmax}_k(\boldsymbol{Mc})_k$

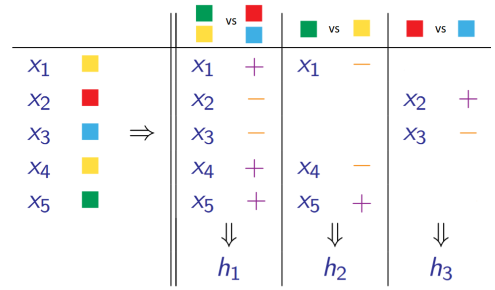
How to design the code $\boldsymbol{M}$?

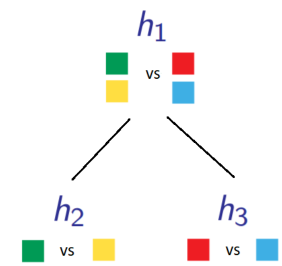
- the more *dissimilar* the codes, the more robust
  - if any two codes are d bits away, then prediction can tolerate about $d/2$ errors
- *random code* is often a good choice

Tree based method

Idea: train ≈ C binary classifiers to learn "**belongs to which half?**".

Training: see pictures

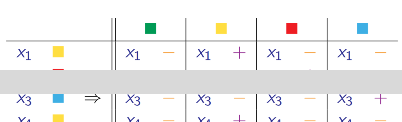




Prediction is also natural, *but is very fast!* (think ImageNet where $C \approx 20K$)

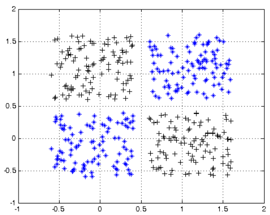
Comparisons

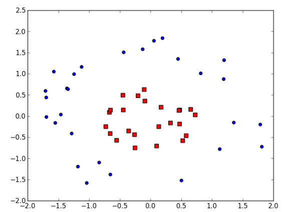
Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO	(C − 1)N	$\mathcal{O}(\mathsf{C}^2)$	can achieve very small training error
ECOC	LN	L	need diversity when designing code
Tree	$\mathcal{O}((\log_2 \mathsf{C})\mathsf{N})$	$\mathcal{O}(\log_2 \mathsf{C})$	good for "extreme classification"



Outline

Linear models are not always adequate





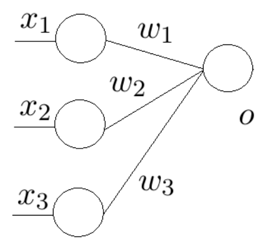
We can use a nonlinear mapping as discussed:

$$\boldsymbol{\phi}(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^{\mathsf{D}} \rightarrow \boldsymbol{z} \in \mathbb{R}^{\mathsf{M}}$$

But what kind of nonlinear mapping ϕ should be used? Can we actually learn this nonlinear mapping?

The most popular nonlinear models nowadays: **neural nets**

Linear model as a one-layer neural net

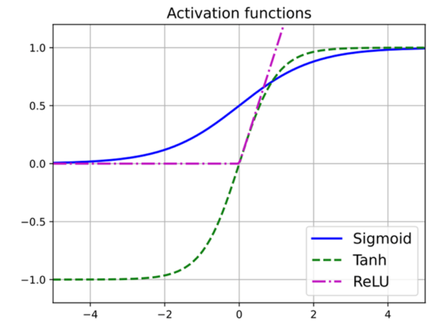


$$o = h(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$$

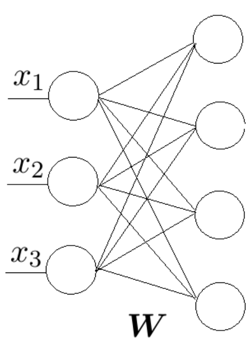
$h(a) = a$ for linear model

To create non-linearity, can use

- Rectified Linear Unit (**ReLU**): $h(a) = \max\{0, a\}$
- sigmoid function: $h(a) = \frac{1}{1+e^{-a}}$
- TanH: $h(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$
- many more



More output nodes

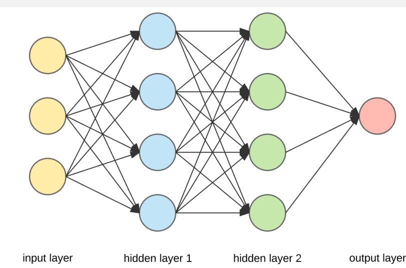


$$\boldsymbol{o} = \boldsymbol{h}(\boldsymbol{W}\boldsymbol{x})$$

$\boldsymbol{W} \in \mathbb{R}^{4\times 3}$, $\boldsymbol{h} : \mathbb{R}^4 \rightarrow \mathbb{R}^4$ so $\boldsymbol{h}(\boldsymbol{a}) = (h_1(a_1), h_2(a_2), h_3(a_3), h_4(a_4))$

Can think of this as a nonlinear mapping: $\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{h}(\boldsymbol{W}\boldsymbol{x})$

More layers



Becomes a network:

- each node is called a **neuron**
- $\boldsymbol{h}$ is called the **activation function**
  - can use $h(a) = 1$ for one neuron in each layer to *incorporate bias term*
  - output neuron can use $h(a) = a$
- #layers refers to #hidden_layers (plus 1 or 2 for input/output layers)
- **deep** neural nets can have many layers and *millions* of parameters
- this is a **feedforward, fully connected** neural net, there are many variants (convolutional nets, recurrent nets, transformers, etc.)

How powerful are neural nets?

Universal approximation theorem (Cybenko, 89; Hornik, 91):

A feedforward neural net with a single hidden layer can approximate any continuous functions.

It might need a huge number of neurons though, and *depth helps!*

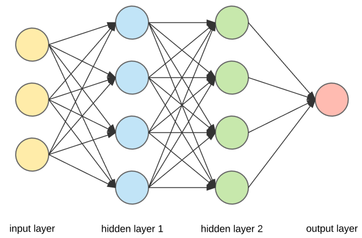
Designing network architecture is important and very complicated

- for feedforward network, need to decide number of hidden layers, number of neurons at each layer, activation functions, etc.

Math formulation

An L-layer neural net can be written as

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{h}_\mathsf{L}\left(\boldsymbol{W}_L \boldsymbol{h}_{\mathsf{L}-1}\left(\boldsymbol{W}_{L-1} \cdots \boldsymbol{h}_1\left(\boldsymbol{W}_1 \boldsymbol{x}\right)\right)\right)$$



To ease notation, for a given input $\boldsymbol{x}$, define recursively

$$\boldsymbol{o}_0 = \boldsymbol{x}, \qquad \boldsymbol{a}_\ell = \boldsymbol{W}_\ell \boldsymbol{o}_{\ell-1}, \qquad \boldsymbol{o}_\ell = \boldsymbol{h}_\ell(\boldsymbol{a}_\ell) \qquad\qquad (\ell = 1, \ldots, \mathsf{L})$$

where

- $\boldsymbol{W}_\ell \in \mathbb{R}^{\mathsf{D}_\ell \times \mathsf{D}_{\ell-1}}$ is the weights between layer $\ell - 1$ and ℓ
- $\mathsf{D}_0 = \mathsf{D}, \mathsf{D}_1, \ldots, \mathsf{D}_\mathsf{L}$ are numbers of neurons at each layer
- $\boldsymbol{a}_\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is input to layer ℓ
- $\boldsymbol{o}_\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is output of layer ℓ
- $\boldsymbol{h}_\ell : \mathbb{R}^{\mathsf{D}_\ell} \to \mathbb{R}^{\mathsf{D}_\ell}$ is activation functions at layer ℓ

Learning the model

No matter how complicated the model is, our goal is the same: minimize

$$F(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_\mathsf{L}) = \frac{1}{N} \sum_{n=1}^{\mathsf{N}} F_n(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_\mathsf{L})$$

where

$$F_n(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_\mathsf{L}) = \begin{cases} \|\boldsymbol{f}(\boldsymbol{x}_n) - \boldsymbol{y}_n\|_2^2 & \text{for regression} \\ \ln\left(1 + \sum_{k \neq y_n} e^{f(\boldsymbol{x}_n)_k - f(\boldsymbol{x}_n)_{y_n}}\right) & \text{for classification} \end{cases}$$

How to optimize such a complicated function?

Same thing: apply **SGD**! even if the model is *nonconvex*.

What is the gradient of this complicated function?

Chain rule is the only secret:

- for a composite function $f(g(w))$

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial w}$$

- for a composite function $f(g_1(w), \ldots, g_d(w))$

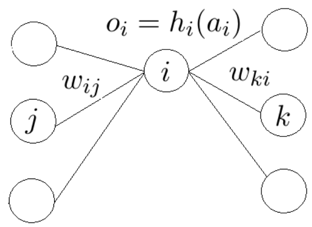
$$\frac{\partial f}{\partial w} = \sum_{i=1}^{d} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial w}$$

  the simplest example $f(g_1(w), g_2(w)) = g_1(w) g_2(w)$

Computing the derivative

Drop the subscript ℓ for layer for simplicity.

Find the **derivative of F_n w.r.t. to w_{ij}**



$$\frac{\partial F_n}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i}\frac{\partial (w_{ij} o_j)}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i} o_j$$

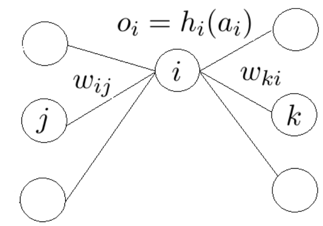
$$\frac{\partial F_n}{\partial a_i} = \frac{\partial F_n}{\partial o_i}\frac{\partial o_i}{\partial a_i} = \left(\sum_k \frac{\partial F_n}{\partial a_k}\frac{\partial a_k}{\partial o_i}\right) h_i'(a_i) = \left(\sum_k \frac{\partial F_n}{\partial a_k} w_{ki}\right) h_i'(a_i)$$

Computing the derivative

Adding the subscript for layer:

$$\frac{\partial F_n}{\partial w_{\ell,ij}} = \frac{\partial F_n}{\partial a_{\ell,i}} o_{\ell-1,j}$$

$$\frac{\partial F_n}{\partial a_{\ell,i}} = \left(\sum_k \frac{\partial F_n}{\partial a_{\ell+1,k}} w_{\ell+1,ki}\right) h_{\ell,i}'(a_{\ell,i})$$



For the last layer, for square loss

$$\frac{\partial F_n}{\partial a_{\mathsf{L},i}} = \frac{\partial (h_{\mathsf{L},i}(a_{\mathsf{L},i}) - y_{n,i})^2}{\partial a_{\mathsf{L},i}} = 2(h_{\mathsf{L},i}(a_{\mathsf{L},i}) - y_{n,i}) h_{\mathsf{L},i}'(a_{\mathsf{L},i})$$

Exercise: try to do it for cross-entropy loss yourself.

Computing the derivative

Using **matrix notation** greatly simplifies presentation and implementation:

$$\frac{\partial F_n}{\partial \boldsymbol{W}_\ell} = \frac{\partial F_n}{\partial \boldsymbol{a}_\ell} \boldsymbol{o}_{\ell-1}^{\mathrm{T}} \in \mathbb{R}^{\mathsf{D}_\ell \times \mathsf{D}_{\ell-1}}$$

$$\frac{\partial F_n}{\partial \boldsymbol{a}_\ell} = \begin{cases} \left(\boldsymbol{W}_{\ell+1}^{\mathrm{T}} \frac{\partial F_n}{\partial \boldsymbol{a}_{\ell+1}}\right) \circ \boldsymbol{h}_\ell'(\boldsymbol{a}_\ell) & \text{if } \ell < \mathsf{L} \\ 2(\boldsymbol{h}_{\mathsf{L}}(\boldsymbol{a}_{\mathsf{L}}) - \boldsymbol{y}_n) \circ \boldsymbol{h}_{\mathsf{L}}'(\boldsymbol{a}_{\mathsf{L}}) & \text{else} \end{cases}$$

where $\boldsymbol{v}_1 \circ \boldsymbol{v}_2 = (v_{11}v_{21}, \cdots, v_{1\mathsf{D}}v_{2\mathsf{D}})$ is the element-wise product (a.k.a. Hadamard product).

Verify yourself!

Putting everything into SGD

The **backpropagation** algorithm (**Backprop**)

Initialize $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}$ randomly. Repeat:

1. randomly pick one data point $n \in [\mathsf{N}]$
2. **forward propagation**: for each layer $\ell = 1, \ldots, \mathsf{L}$
   - compute $\boldsymbol{a}_\ell = \boldsymbol{W}_\ell \boldsymbol{o}_{\ell-1}$ and $\boldsymbol{o}_\ell = \boldsymbol{h}_\ell(\boldsymbol{a}_\ell)$ $\quad(\boldsymbol{o}_0 = \boldsymbol{x}_n)$
3. **backward propagation**: for each $\ell = L, \ldots, 1$
   - compute

$$\frac{\partial F_n}{\partial \boldsymbol{a}_\ell} = \begin{cases} \left(\boldsymbol{W}_{\ell+1}^{\mathrm{T}} \frac{\partial F_n}{\partial \boldsymbol{a}_{\ell+1}}\right) \circ \boldsymbol{h}_\ell'(\boldsymbol{a}_\ell) & \text{if } \ell < \mathsf{L} \\ 2(\boldsymbol{h}_{\mathsf{L}}(\boldsymbol{a}_{\mathsf{L}}) - \boldsymbol{y}_n) \circ \boldsymbol{h}_{\mathsf{L}}'(\boldsymbol{a}_{\mathsf{L}}) & \text{else} \end{cases}$$

   - update weights

$$\boldsymbol{W}_\ell \leftarrow \boldsymbol{W}_\ell - \eta \frac{\partial F_n}{\partial \boldsymbol{W}_\ell} = \boldsymbol{W}_\ell - \eta \frac{\partial F_n}{\partial \boldsymbol{a}_\ell} \boldsymbol{o}_{\ell-1}^{\mathrm{T}}$$

(Important: *should $\boldsymbol{W}_\ell$ be overwritten immediately in the last step?*)

Important tricks to optimize neural nets

Many important tricks on top on Backprop

- **mini-batch**: randomly sample a batch of examples to form a stochastic gradient (common batch size: 32, 64, 128, etc.)
- **batch normalization**: normalize the inputs of each neuron over the mini-batch (to zero-mean and one-variance; *c.f.* Lec 1)
- **adaptive learning rate**: scale the learning rate of each parameter based on some moving average of the magnitude of the gradients
- **momentum**: make use of previous gradients (taking inspiration from physics)
- $\cdots$

SGD with momentum (a simple version)

Initialize $\boldsymbol{w}_0$ and **velocity** $\boldsymbol{v} = \mathbf{0}$

For $t = 1, 2, \ldots$

- form a stochastic gradient $\boldsymbol{g}_t$
- update velocity $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} + \boldsymbol{g}_t$ for some discount factor $\alpha \in (0, 1)$
- update weight $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \eta \boldsymbol{v}$

Updates for first few rounds:

- $\boldsymbol{w}_1 = \boldsymbol{w}_0 - \eta \boldsymbol{g}_1$
- $\boldsymbol{w}_2 = \boldsymbol{w}_1 - \alpha\eta \boldsymbol{g}_1 - \eta \boldsymbol{g}_2$
- $\boldsymbol{w}_3 = \boldsymbol{w}_2 - \alpha^2\eta \boldsymbol{g}_1 - \alpha\eta \boldsymbol{g}_2 - \eta \boldsymbol{g}_3$
- $\cdots$

Adam (most popular) $\approx$ SGD + adaptive learning rate + momentum

Overfitting

Overfitting is very likely since neural nets are too powerful.

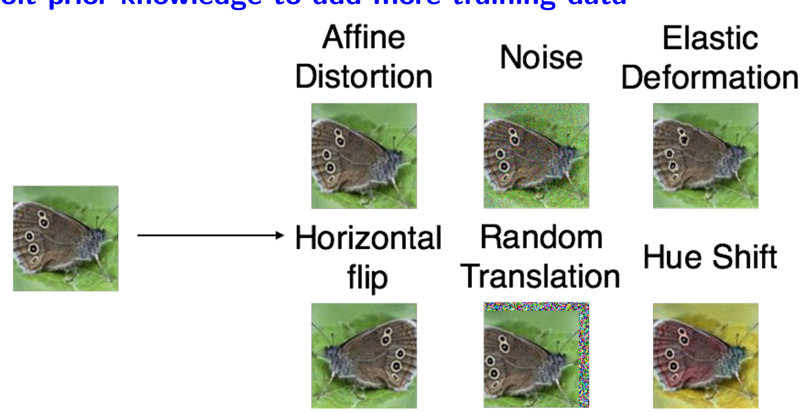
Methods to overcome overfitting:

- data augmentation
- regularization
- dropout
- early stopping
- $\cdots$

Data augmentation

Data: the more the better. How do we get more data?

Exploit prior knowledge to add more training data



Regularization

L2 regularization: minimize

$$F'(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_\mathsf{L}) = F(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_\mathsf{L}) + \lambda \sum_{\ell=1}^{\mathsf{L}} \|\boldsymbol{W}_\ell\|_2^2$$

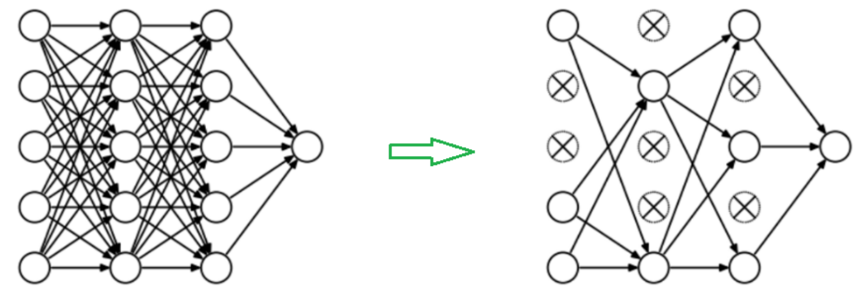
Simple change to the gradient:

$$\frac{\partial F'}{\partial w_{ij}} = \frac{\partial F}{\partial w_{ij}} + 2\lambda w_{ij}$$

Introduce *weight decaying effect*

Dropout

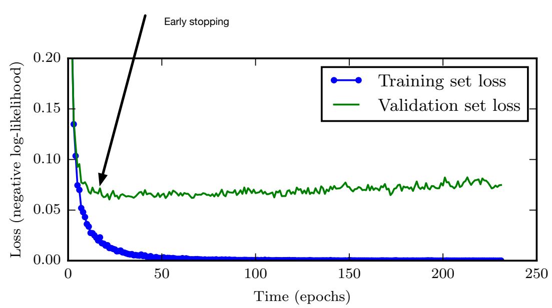
Independently delete each neuron with a fixed probability (say 0.5), during each iteration of Backprop (only for training, not for testing)



Very effective, makes training faster as well

Early stopping

Stop training when the performance on validation set stops improving



Conclusions for neural nets

Deep neural networks

- are hugely popular, achieving *best performance* on many problems
- do need *a lot of data* to work well
- take *a lot of time* to train (need GPUs for massive parallel computing)
- take some work to select architecture and hyperparameters
- are still not well understood in theory