

# CSCI567 Machine Learning (Fall 2021)

Prof. Haipeng Luo

U of Southern California

Sep 16, 2021

# Administration

HW1 is being graded. Will discuss solutions today.

HW2 will be released after this lecture. Due on 9/28.

# Outline

- 1 Review of Last Lecture
- 2 Multiclass Classification
- 3 Neural Nets

# Outline

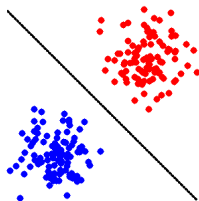
- 1 Review of Last Lecture
- 2 Multiclass Classification
- 3 Neural Nets

# Linear classifiers

Linear models for **binary** classification:

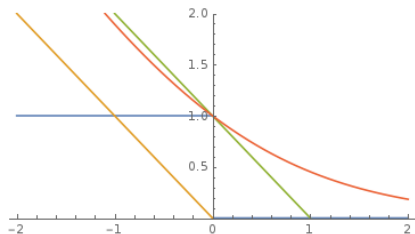
Step 1. Model is the set of **separating hyperplanes**

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



# 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):

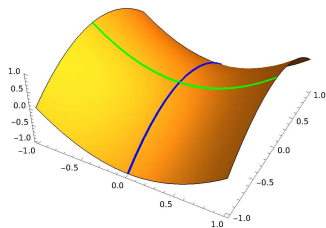
$$\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w} \in \mathbb{R}^D} F(\mathbf{w}) = \operatorname{argmin}_{\mathbf{w} \in \mathbb{R}^D} \frac{1}{N} \sum_{n=1}^N \ell(y_n \mathbf{w}^T \mathbf{x}_n)$$

using

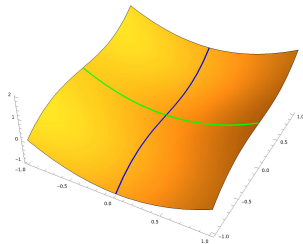
- **GD:**  $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla F(\mathbf{w})$
- **SGD:**  $\mathbf{w} \leftarrow \mathbf{w} - \eta \tilde{\nabla} F(\mathbf{w})$   $(\mathbb{E}[\tilde{\nabla} F(\mathbf{w})] = \nabla F(\mathbf{w}))$
- **Newton:**  $\mathbf{w} \leftarrow \mathbf{w} - (\nabla^2 F(\mathbf{w}))^{-1}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

# 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



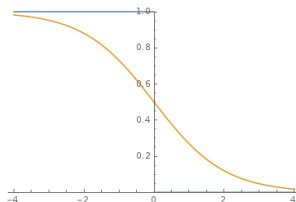
# Perceptron and logistic regression

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

Repeat:

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

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



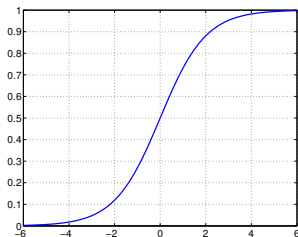
# A Probabilistic view of logistic regression

## Minimizing logistic loss = MLE for the sigmoid model

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

where

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



# Outline

- 1 Review of Last Lecture
- 2 Multiclass Classification
  - Multinomial logistic regression
  - Reduction to binary classification
- 3 Neural Nets

# Classification

Recall the setup:

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

## Examples:

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

**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  $\{1, 2\}$ )

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

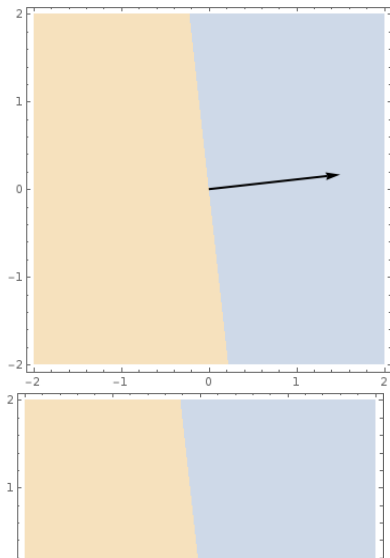
can be written as

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

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

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

# Linear models: from binary to multiclass



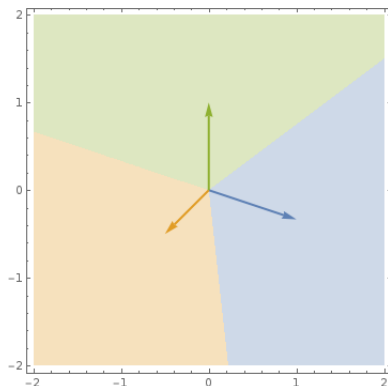
$$\mathbf{w} = \left(\frac{3}{2}, \frac{1}{6}\right) = \mathbf{w}_1 - \mathbf{w}_2$$

$$\mathbf{w}_1 = \left(1, -\frac{1}{3}\right)$$

$$\mathbf{w}_2 = \left(-\frac{1}{2}, -\frac{1}{2}\right)$$

- Blue class:

# Linear models: from binary to multiclass



$$w_1 = (1, -\frac{1}{3})$$

$$w_2 = (-\frac{1}{2}, -\frac{1}{2})$$

$$w_3 = (0, 1)$$

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

# Linear models for multiclass classification

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

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  $\mathbf{w} = \mathbf{w}_1 - \mathbf{w}_2$ :

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

Naturally, for multiclass:

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

This is called the *softmax function*.

# Applying MLE again

Maximize probability of seeing labels  $y_1, \dots, y_N$  given  $\mathbf{x}_1, \dots, \mathbf{x}_N$

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

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

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

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

When  $C = 2$ , this is the same as binary logistic loss.

## Step 3: Optimization

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

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

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

If  $k \neq y_n$ :

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

else:

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

# SGD for multinomial logistic regression

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

- 1 pick  $n \in [N]$  uniformly at random
- 2 update the parameters

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

Think about why the algorithm makes sense intuitively.

# A note on prediction

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

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

In either case, **(expected) mistake is bounded by logistic loss**

- deterministic

$$\mathbb{I}[f(\mathbf{x}) \neq y] \leq \log_2 \left( 1 + \sum_{k \neq y} e^{(\mathbf{w}_k - \mathbf{w}_y)^T \mathbf{x}} \right)$$

- randomized

$$\mathbb{E} [\mathbb{I}[f(\mathbf{x}) \neq y]] = 1 - \mathbb{P}(y \mid \mathbf{x}; \mathbf{W}) \leq -\ln \mathbb{P}(y \mid \mathbf{x}; \mathbf{W})$$

# 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  $k \in [C]$ ,

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

					
$x_1$		$x_1$ —	$x_1$ +	$x_1$ —	$x_1$ —
$x_2$		$x_2$ —	$x_2$ —	$x_2$ +	$x_2$ —
$x_3$		$x_3$ —	$x_3$ —	$x_3$ —	$x_3$ +
$x_4$		$x_4$ —	$x_4$ +	$x_4$ —	$x_4$ —
$x_5$		$x_5$ +	$x_5$ —	$x_5$ —	$x_5$ —
	$\Rightarrow$	$\Downarrow$ $h_1$	$\Downarrow$ $h_2$	$\Downarrow$ $h_3$	$\Downarrow$ $h_4$

# One-versus-all (OvA)

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

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

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. ■
		vs.	vs.	vs.	vs.	vs.	vs.
$x_1$	■	$x_1$ —			$x_1$ —		$x_1$ —
$x_2$	■		$x_2$ —	$x_2$ +			$x_2$ +
$x_3$	■			$x_3$ —	$x_3$ +	$x_3$ —	
$x_4$	■	$x_4$ —			$x_4$ —		$x_4$ —
$x_5$	■	$x_5$ +	$x_5$ +			$x_5$ +	
	⇒	⇓	⇓	⇓	⇓	⇓	⇓
		$h_{(1,2)}$	$h_{(1,3)}$	$h_{(3,4)}$	$h_{(4,2)}$	$h_{(1,4)}$	$h_{(3,2)}$

# One-versus-one (OvO)

Prediction: for a new example  $x$

- 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  $\mathbf{M} \in \{-1, +1\}^{C \times L}$ , train  $L$  binary classifiers to learn “**is bit  $b$  on or off**”.

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

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

$\mathbf{M}$	1	2	3	4	5
	+	-	+	-	+
	-	-	+	+	+
	+	+	-	-	-
	+	+	+	+	-

	1	2	3	4	5
$x_1$ 	$x_1$ -	$x_1$ -	$x_1$ +	$x_1$ +	$x_1$ +
$x_2$ 	$x_2$ +	$x_2$ +	$x_2$ -	$x_2$ -	$x_2$ -
$x_3$ 	$x_3$ +	$x_3$ +	$x_3$ +	$x_3$ +	$x_3$ -
$x_4$ 	$x_4$ -	$x_4$ -	$x_4$ +	$x_4$ +	$x_4$ +
$x_5$ 	$x_5$ +	$x_5$ -	$x_5$ +	$x_5$ -	$x_5$ +
	$\Downarrow$ $h_1$	$\Downarrow$ $h_2$	$\Downarrow$ $h_3$	$\Downarrow$ $h_4$	$\Downarrow$ $h_5$

# Error-correcting output codes (ECOC)

Prediction: for a new example  $x$

- compute the **predicted code**  $c = (h_1(x), \dots, h_L(x))^T$
- predict the class with the **most similar code**:  $k = \operatorname{argmax}_k (Mc)_k$


















How to design the code  $M$ ?

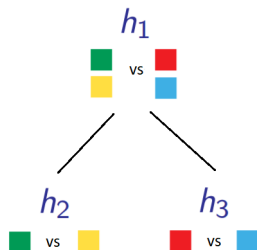
- the more *dissimilar* the codes, the more robust
  - if any two codes are  $d$  bits away, then prediction can tolerate about  $d/2$  errors
- *random code* is often a good choice

# Tree based method

Idea: train  $\approx C$  binary classifiers to learn “**belongs to which half?**”.

Training: see pictures

		 vs   	 vs   	 vs   
$x_1$		$x_1$ +	$x_1$ —	
$x_2$		$x_2$ —		$x_2$ +
$x_3$		$x_3$ —		$x_3$ —
$x_4$		$x_4$ +	$x_4$ —	
$x_5$		$x_5$ +	$x_5$ +	
		$\Downarrow$ $h_1$	$\Downarrow$ $h_2$	$\Downarrow$ $h_3$



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

# Comparisons

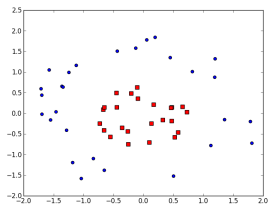
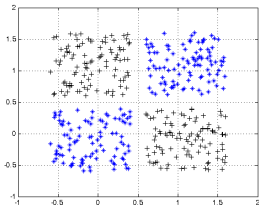
Reduction	training time	prediction time	remark
OvA	CN	C	not robust
OvO	$(C - 1)N$	$\mathcal{O}(C^2)$	can achieve very small training error
ECOC	LN	L	need diversity when designing code
Tree	$\mathcal{O}((\log_2 C)N)$	$\mathcal{O}(\log_2 C)$	good for “extreme classification”



# Outline

- 1 Review of Last Lecture
- 2 Multiclass Classification
- 3 Neural Nets
  - Definition
  - Backpropagation
  - Preventing overfitting

# Linear models are not always adequate



We can use a nonlinear mapping as discussed:

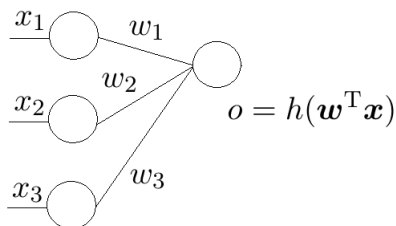
$$\phi(x) : x \in \mathbb{R}^D \rightarrow z \in \mathbb{R}^M$$

*But what kind of nonlinear mapping  $\phi$  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

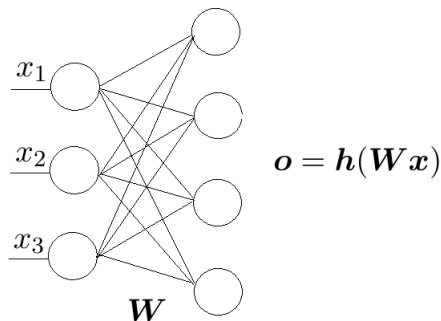


$$h(a) = a \text{ for linear model}$$

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



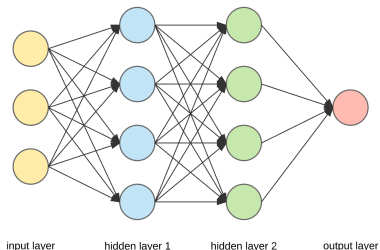
$\mathbf{W} \in \mathbb{R}^{4 \times 3}$ ,  $\mathbf{h} : \mathbb{R}^4 \rightarrow \mathbb{R}^4$  so  $\mathbf{h}(\mathbf{a}) = (h_1(a_1), h_2(a_2), h_3(a_3), h_4(a_4))$

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

# More layers

Becomes a network:

- each node is called a **neuron**
- $h$  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, residual nets, recurrent nets, 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!*

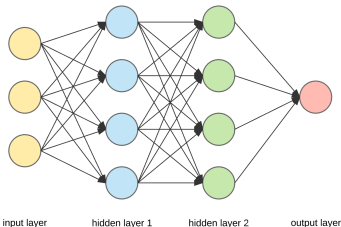
Designing network architecture is important and very complicated

- 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

$$f(x) = h_L(W_L h_{L-1}(W_{L-1} \cdots h_1(W_1 x)))$$



To ease notation, for a given input  $x$ , define recursively

$$o_0 = x, \quad a_\ell = W_\ell o_{\ell-1}, \quad o_\ell = h_\ell(a_\ell) \quad (\ell = 1, \dots, L)$$

where

- $W_\ell \in \mathbb{R}^{D_\ell \times D_{\ell-1}}$  is the weights between layer  $\ell - 1$  and  $\ell$
- $D_0 = D, D_1, \dots, D_L$  are numbers of neurons at each layer
- $a_\ell \in \mathbb{R}^{D_\ell}$  is input to layer  $\ell$
- $o_\ell \in \mathbb{R}^{D_\ell}$  is output of layer  $\ell$
- $h_\ell : \mathbb{R}^{D_\ell} \rightarrow \mathbb{R}^{D_\ell}$  is activation functions at layer  $\ell$

# Learning the model

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

$$F(\mathbf{W}_1, \dots, \mathbf{W}_L) = \frac{1}{N} \sum_{n=1}^N F_n(\mathbf{W}_1, \dots, \mathbf{W}_L)$$

where

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

# 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  $f(g_1(w), \dots, g_d(w))$

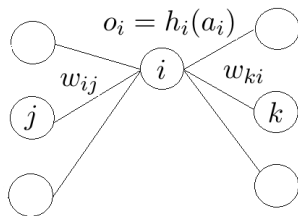
$$\frac{\partial f}{\partial w} = \sum_{i=1}^d \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial w}$$

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

# Computing the derivative

Drop the subscript  $\ell$  for layer for simplicity.

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



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

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

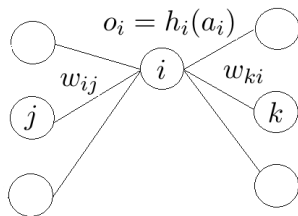


# Computing the derivative

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

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

**Exercise:** try to do it for logistic loss yourself.

# Computing the derivative

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

$$\frac{\partial F_n}{\partial \mathbf{W}_\ell} = \frac{\partial F_n}{\partial \mathbf{a}_\ell} \mathbf{o}_{\ell-1}^T \in \mathbb{R}^{D_\ell \times D_{\ell-1}}$$

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

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

Verify yourself!

# Putting everything into SGD

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

Initialize  $\mathbf{W}_1, \dots, \mathbf{W}_L$  randomly. Repeat:

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

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

- update weights

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

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

# More tricks to optimize neural nets

Many variants based 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)
- **momentum**: make use of previous gradients (taking inspiration from physics)
- ...

# SGD with momentum (a simple version)

Initialize  $w_0$  and **velocity**  $v = 0$

For  $t = 1, 2, \dots$

- form a stochastic gradient  $g_t$
- update velocity  $v \leftarrow \alpha v + g_t$  for some discount factor  $\alpha \in (0, 1)$
- update weight  $w_t \leftarrow w_{t-1} - \eta v$

Updates for first few rounds:

- $w_1 = w_0 - \eta g_1$
- $w_2 = w_1 - \alpha \eta g_1 - \eta g_2$
- $w_3 = w_2 - \alpha^2 \eta g_1 - \alpha \eta g_2 - \eta g_3$
- $\dots$

# Overfitting

**Overfitting is very likely** since neural nets are too powerful.

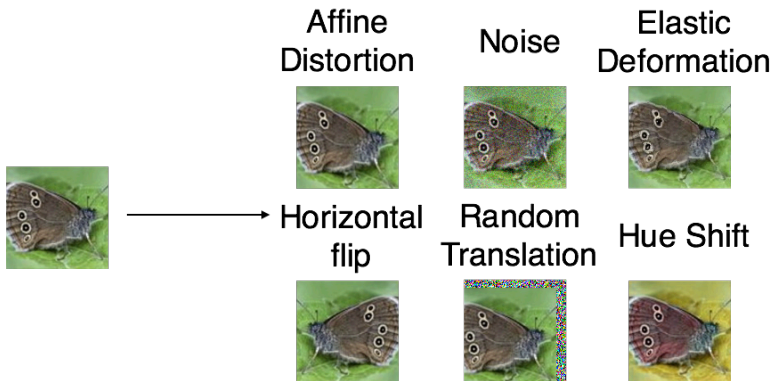
Methods to overcome overfitting:

- data augmentation
- regularization
- dropout
- early stopping
- ...

# 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

$$F'(\mathbf{W}_1, \dots, \mathbf{W}_L) = F(\mathbf{W}_1, \dots, \mathbf{W}_L) + \lambda \sum_{\ell=1}^L \|\mathbf{W}_\ell\|_2^2$$

Simple change to the gradient:

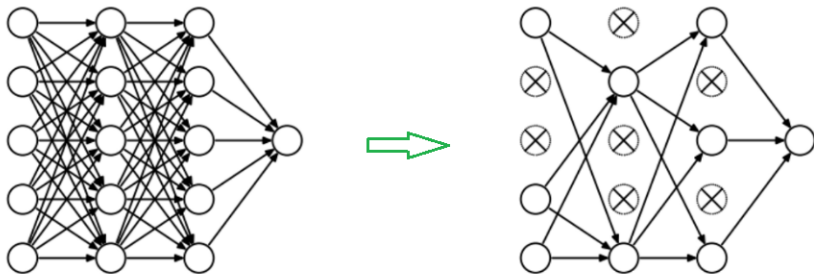
$$\frac{\partial F'}{\partial w_{ij}} = \frac{\partial F}{\partial w_{ij}} + 2\lambda w_{ij}$$

Introduce *weight decaying effect*



# Dropout

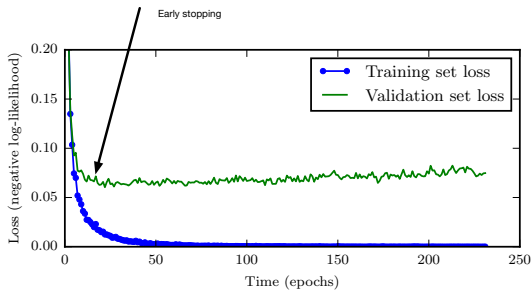
**Independently delete each neuron** with a fixed probability (say 0.5), during each iteration of Backprop (only for training, not for testing)



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