CSCI567 Machine Learning (Fall 2023)

Prof. Dani Yogatama
Slide Deck from Prof. Haipeng Luo

University of Southern California

Sep 15, 2023

Outline

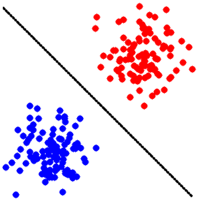
Outline

Linear classifiers

Linear models for **binary** classification:

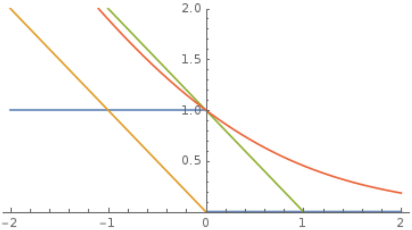
Step 1. Model is the set of **separating hyperplanes**

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



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

Convergence guarantees of GD/SGD

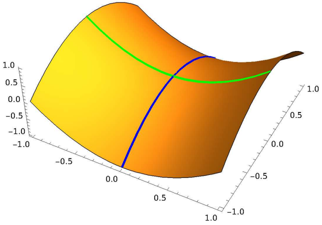
- GD/SGD converges to a stationary point

Convergence guarantees of GD/SGD

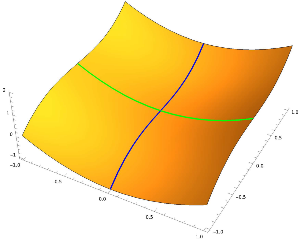
- GD/SGD converges to a stationary point
- for convex objectives, this is all we need

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

Repeat:

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

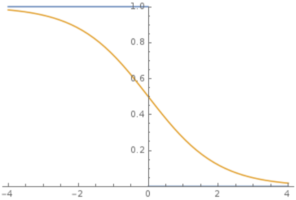
Perceptron and logistic regression

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

Repeat:

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

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



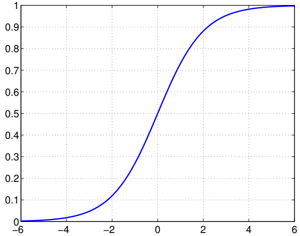
A Probabilistic view of logistic regression

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 $f : \mathbb{R}^{\mathsf{D}} \rightarrow [\mathsf{C}]$

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 ($\mathsf{C} \approx 20K$)

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?*

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 $\{1, 2\}$)

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

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

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \geq \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} \\ 2 & \text{if } \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} > \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \end{cases}$$

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

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. $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$

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 $\{1, 2\}$)

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

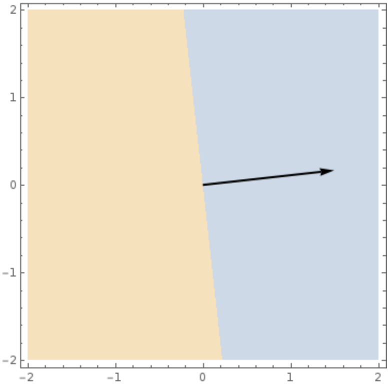
can be written as

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

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**.

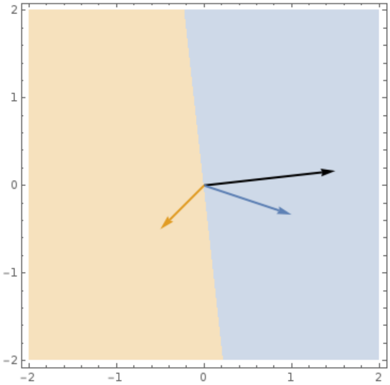
Linear models: from binary to multiclass



$$\boldsymbol{w} = \left(\tfrac{3}{2}, \tfrac{1}{6}\right)$$

- Blue class: $\{\boldsymbol{x} : \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \geq 0\}$
- Orange class: $\{\boldsymbol{x} : \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} < 0\}$

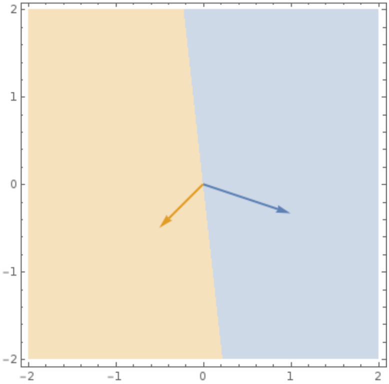
Linear models: from binary to multiclass



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

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

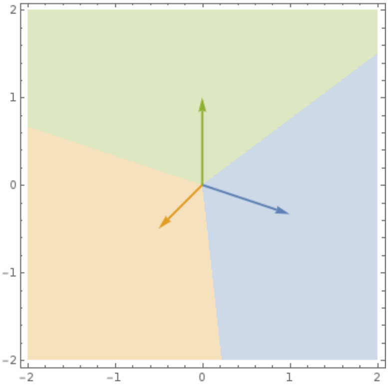
Linear models: from binary to multiclass



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

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

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

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

Linear models for multiclass classification

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

Linear models for multiclass classification

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

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

Linear models for multiclass classification

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

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

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

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:

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

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 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\mathsf{N}$

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

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

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

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

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

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*.

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

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

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

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

If $k \neq y_n$:

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

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

If $k \neq y_n$:

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

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 C $\times$ 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:

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

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 C $\times$ 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 $\boldsymbol{W} = \mathbf{0}$ (or randomly). Repeat:

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

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

SGD for multinomial logistic regression

Initialize $\boldsymbol{W} = \mathbf{0}$ (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 $\operatorname{argmax}_{k\in[\mathsf{C}]}\ \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}$

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 $\mathbb{P}(k \mid \boldsymbol{x}; \boldsymbol{W}) \propto e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}}$

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 $\mathbb{P}(k \mid \boldsymbol{x}; \boldsymbol{W}) \propto e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}$

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

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 $\mathbb{P}(k \mid \boldsymbol{x}; \boldsymbol{W}) \propto e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}$

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

- deterministic

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

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 $\mathbb{P}(k \mid \boldsymbol{x};\boldsymbol{W}) \propto e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}$

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

- deterministic

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

- randomized

$$\mathbb{E}\left[\mathbb{I}[f(\boldsymbol{x}) \neq y]\right]$$

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 $\mathbb{P}(k \mid \boldsymbol{x}; \boldsymbol{W}) \propto e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}$

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

- deterministic

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

- randomized

$$\mathbb{E}\left[\mathbb{I}[f(\boldsymbol{x}) \neq y]\right] = 1 - \mathbb{P}(y \mid \boldsymbol{x}; \boldsymbol{W})$$

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 $\mathbb{P}(k \mid \boldsymbol{x}; \boldsymbol{W}) \propto e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}}$

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

- deterministic

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

- randomized

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

Reduce multiclass to binary

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

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*?

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.

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

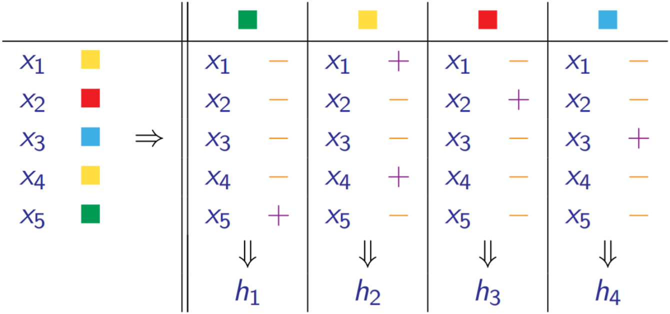
One-versus-all (OvA)

(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. $h_k(\boldsymbol{x})$)

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. $h_k(\boldsymbol{x}) = +1$.

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

One-versus-one (OvO)

(picture credit: link)

Idea: train $\binom{\mathsf{C}}{2}$ 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

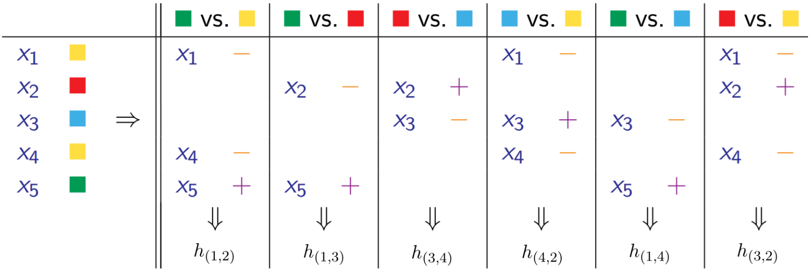
One-versus-one (OvO)

(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



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'**

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)

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**".

M	1	2	3	4	5
■	+	−	+	−	+
■	−	−	+	+	+
■	+	+	−	−	−
■	+	+	+	+	−

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** $\boldsymbol{c} = (h_1(\boldsymbol{x}), \ldots, h_{\mathsf{L}}(\boldsymbol{x}))^{\mathrm{T}}$

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**: $k = \operatorname{argmax}_k(\boldsymbol{M}\boldsymbol{c})_k$

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**: $k = \mathrm{argmax}_k(\boldsymbol{M}\boldsymbol{c})_k$

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

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

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**: $k = \mathrm{argmax}_k(\boldsymbol{M}\boldsymbol{c})_k$

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

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**: $k = \mathrm{argmax}_k(\boldsymbol{M}\boldsymbol{c})_k$

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

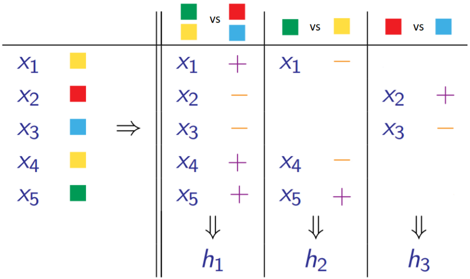
Tree based method

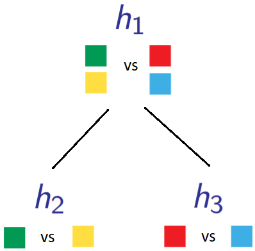
Idea: train $\approx C$ binary classifiers to learn "**belongs to which half?**".

Tree based method

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

Training: see pictures

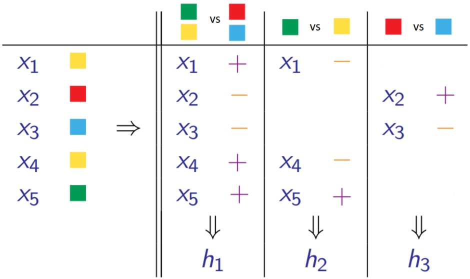


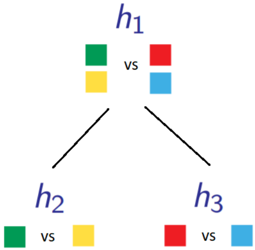


Tree based method

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

Training: see pictures



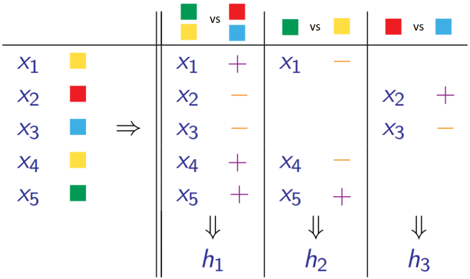


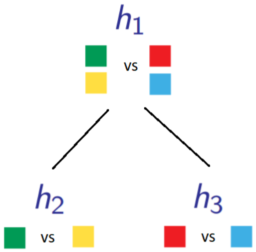
Prediction is also natural,

Tree based method

Idea: train $\approx$ 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**

training time: how many training points are created

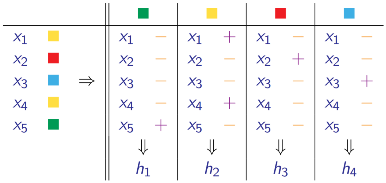
prediction time: how many binary predictions are made

Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA			

training time: how many training points are created

prediction time: how many binary predictions are made

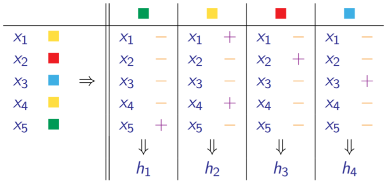


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN		

training time: how many training points are created

prediction time: how many binary predictions are made

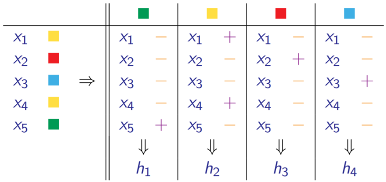


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	

training time: how many training points are created

prediction time: how many binary predictions are made

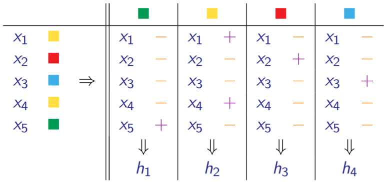


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust

training time: how many training points are created

prediction time: how many binary predictions are made

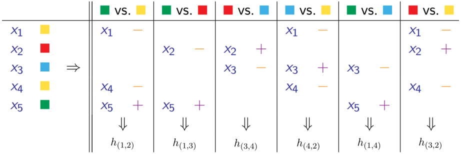


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO			

training time: how many training points are created

prediction time: how many binary predictions are made

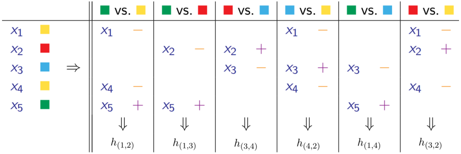


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO	$(C-1)N$		

training time: how many training points are created

prediction time: how many binary predictions are made

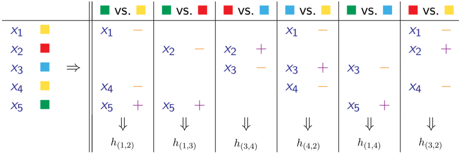


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO	$(C-1)N$	$\mathcal{O}(C^2)$	

training time: how many training points are created

prediction time: how many binary predictions are made

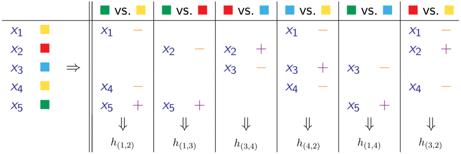


Comparisons

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

training time: how many training points are created

prediction time: how many binary predictions are made



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			

training time: how many training points are created

prediction time: how many binary predictions are made

			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	

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		

training time: how many training points are created

prediction time: how many binary predictions are made

			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	

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	

training time: how many training points are created

prediction time: how many binary predictions are made

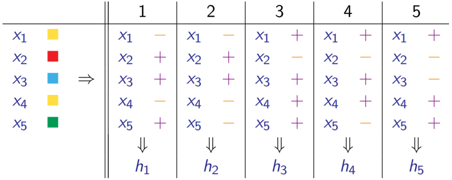
			1		2		3		4		5	
x_1	■		x_1	$-$	x_1	$-$	x_1	$+$	x_1	$+$	x_1	$+$
x_2	■		x_2	$+$	x_2	$+$	x_2	$-$	x_2	$-$	x_2	$-$
x_3	■	$\Rightarrow$	x_3	$+$	x_3	$+$	x_3	$+$	x_3	$+$	x_3	$-$
x_4	■		x_4	$-$	x_4	$-$	x_4	$+$	x_4	$+$	x_4	$+$
x_5	■		x_5	$+$	x_5	$-$	x_5	$+$	x_5	$-$	x_5	$+$
			$\Downarrow$		$\Downarrow$		$\Downarrow$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

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

training time: how many training points are created

prediction time: how many binary predictions are made

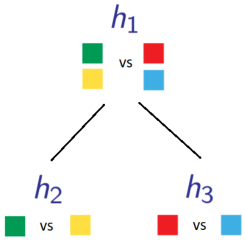


Comparisons

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			

training time: how many training points are created

prediction time: how many binary predictions are made

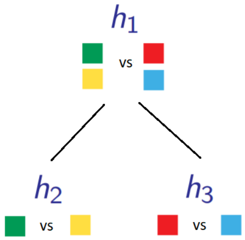


Comparisons

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	$\mathcal{O}((\log_2 C)N)$		

training time: how many training points are created

prediction time: how many binary predictions are made

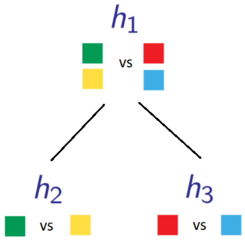


Comparisons

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	$\mathcal{O}((\log_2 C)N)$	$\mathcal{O}(\log_2 C)$	

training time: how many training points are created

prediction time: how many binary predictions are made

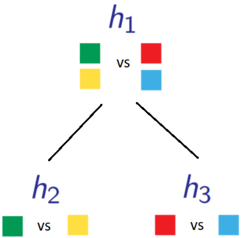


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO	$(\mathsf{C}-1)\mathsf{N}$	$\mathcal{O}(\mathsf{C}^2)$	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"

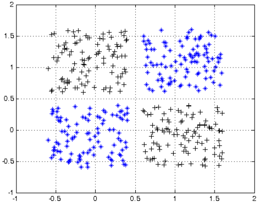
training time: how many training points are created

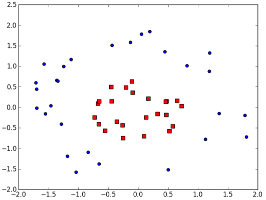
prediction time: how many binary predictions are made



Outline

Linear models are not always adequate

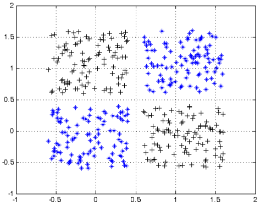


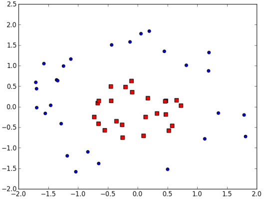


We can use a nonlinear mapping as discussed:

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

Linear models are not always adequate



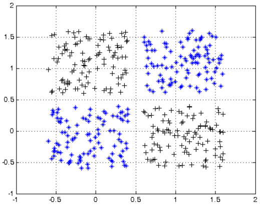


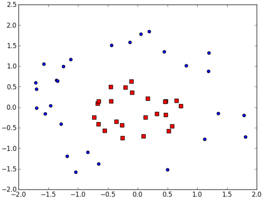
We can use a nonlinear mapping as discussed:

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

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

Linear models are not always adequate





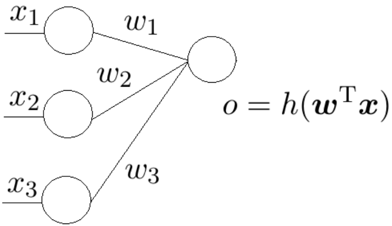
We can use a nonlinear mapping as discussed:

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

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

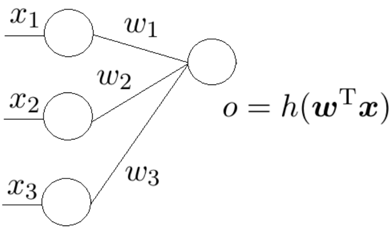
THE most popular nonlinear models nowadays: **neural nets**

Linear model as a one-layer neural net



$h(a) = a$ for linear model

Linear model as a one-layer neural net

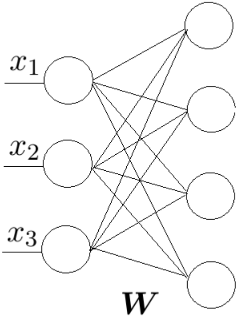


$h(a) = a$ for linear model

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

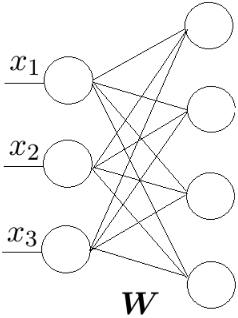
More output nodes



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

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

More output nodes



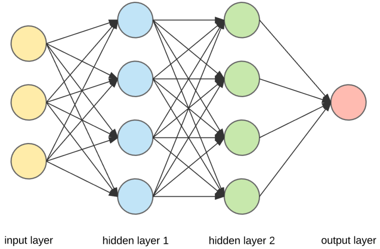
$$\boldsymbol{o} = \boldsymbol{h}(\boldsymbol{W}\boldsymbol{x})$$

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

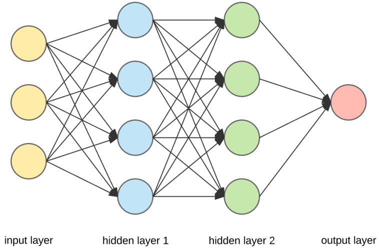
Can think of this as a nonlinear mapping: $\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{h}(\boldsymbol{W}\boldsymbol{x})$

More layers

Becomes a network:



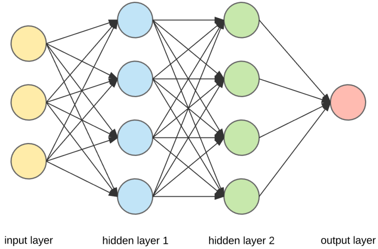
More layers



Becomes a network:

- each node is called a **neuron**

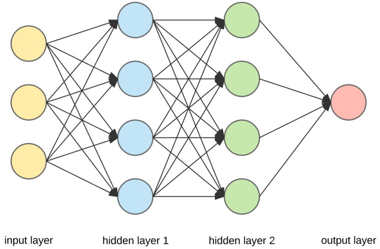
More layers



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 $h(a) = a$

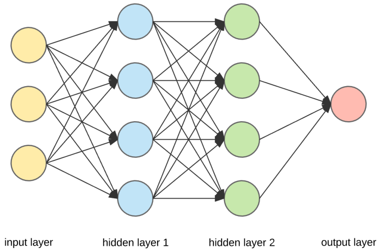
More layers



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)

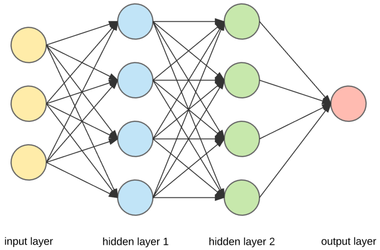
More layers



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

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, 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.

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!*

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!*

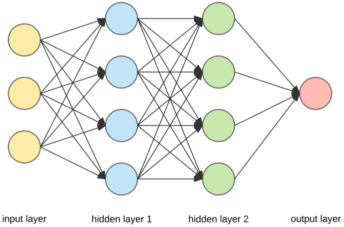
Designing network architecture is important and very complicated

- 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

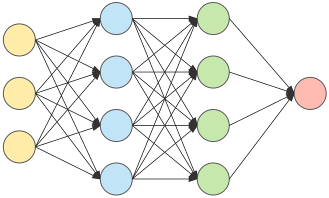
$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{h}_{\mathsf{L}}\left(\boldsymbol{W}_L \boldsymbol{h}_{\mathsf{L}-1}\left(\boldsymbol{W}_{L-1} \cdots \boldsymbol{h}_1\left(\boldsymbol{W}_1 \boldsymbol{x}\right)\right)\right)$$



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

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

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*.

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?

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 $f(g(w))$

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

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 $f(g_1(w), \ldots, g_d(w))$

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

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 $f(g_1(w), \ldots, g_d(w))$

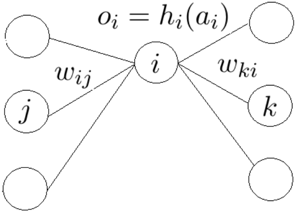
$$\frac{\partial f}{\partial w} = \sum_{i=1}^{d} \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial w}$$

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

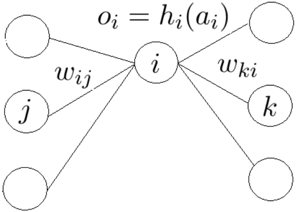
Computing the derivative

Drop the subscript ℓ for layer for simplicity.

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



Computing the derivative

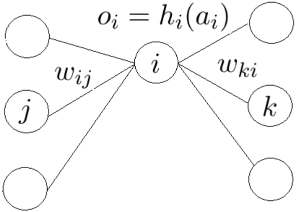


Drop the subscript ℓ for layer for simplicity.

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

$$\frac{\partial F_n}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i} \frac{\partial a_i}{\partial w_{ij}}$$

Computing the derivative



Drop the subscript ℓ for layer for simplicity.

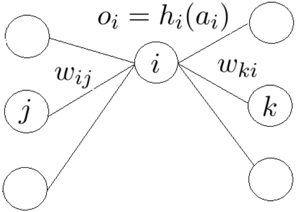
Find the **derivative of F_n w.r.t. to w_{ij}**

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

Computing the derivative

Drop the subscript ℓ for layer for simplicity.

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

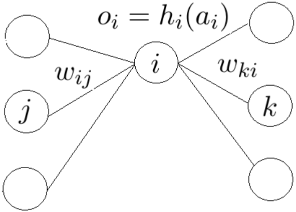


$$\frac{\partial F_n}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i}\frac{\partial (w_{ij} o_j)}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i} o_j$$

Computing the derivative

Drop the subscript ℓ for layer for simplicity.

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



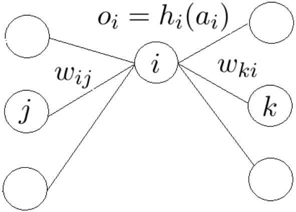
$$\frac{\partial F_n}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i}\frac{\partial (w_{ij} o_j)}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i} o_j$$

$$\frac{\partial F_n}{\partial a_i} = \frac{\partial F_n}{\partial o_i}\frac{\partial o_i}{\partial a_i}$$

Computing the derivative

Drop the subscript ℓ for layer for simplicity.

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



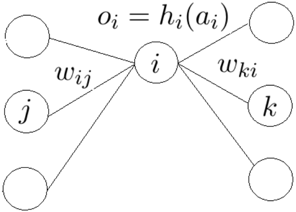
$$\frac{\partial F_n}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i}\frac{\partial (w_{ij}o_j)}{\partial w_{ij}} = \frac{\partial F_n}{\partial a_i} o_j$$

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

Computing the derivative

Drop the subscript ℓ for layer for simplicity.

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



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

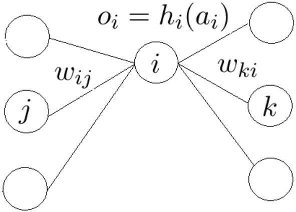
$$\frac{\partial F_n}{\partial a_i} = \frac{\partial F_n}{\partial o_i}\frac{\partial o_i}{\partial a_i} = \left(\sum_k \frac{\partial F_n}{\partial a_k}\frac{\partial a_k}{\partial o_i}\right) h_i'(a_i) = \left(\sum_k \frac{\partial F_n}{\partial a_k} w_{ki}\right) h_i'(a_i)$$

Computing the derivative

Adding the subscript for layer:

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

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

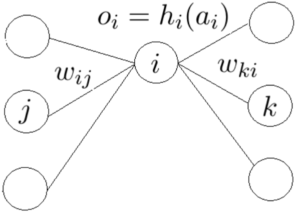


Computing the derivative

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

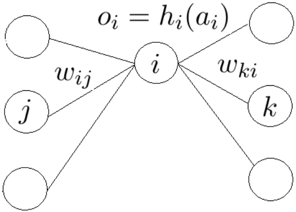
$$\frac{\partial F_n}{\partial a_{\mathsf{L},i}} = \frac{\partial (h_{\mathsf{L},i}(a_{\mathsf{L},i}) - y_{n,i})^2}{\partial a_{\mathsf{L},i}}$$

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

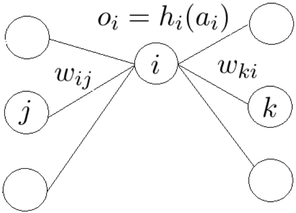
$$\frac{\partial F_n}{\partial a_{\mathsf{L},i}} = \frac{\partial (h_{\mathsf{L},i}(a_{\mathsf{L},i}) - y_{n,i})^2}{\partial a_{\mathsf{L},i}} = 2(h_{\mathsf{L},i}(a_{\mathsf{L},i}) - y_{n,i}) h'_{\mathsf{L},i}(a_{\mathsf{L},i})$$

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 logistic 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.

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 $n \in [\mathsf{N}]$

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 $\boldsymbol{o}_\ell = \boldsymbol{h}_\ell(\boldsymbol{a}_\ell)$ $\quad (\boldsymbol{o}_0 = \boldsymbol{x}_n)$

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 $\boldsymbol{o}_\ell = \boldsymbol{h}_\ell(\boldsymbol{a}_\ell)$ $\qquad (\boldsymbol{o}_0 = \boldsymbol{x}_n)$
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

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

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 $\boldsymbol{o}_\ell = \boldsymbol{h}_\ell(\boldsymbol{a}_\ell)$ $\qquad (\boldsymbol{o}_0 = \boldsymbol{x}_n)$
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?*)

More tricks to optimize neural nets

Many variants based on Backprop

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.)

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)

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

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 $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \eta\boldsymbol{v}$

SGD with momentum (a simple version)

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

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:

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

Overfitting

Overfitting is very likely since neural nets are too powerful.

Methods to overcome overfitting:

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

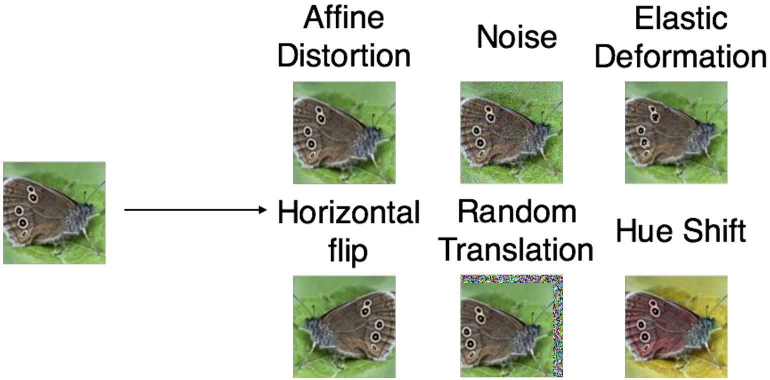
Data augmentation

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

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

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

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:

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

Regularization

L2 regularization: minimize

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

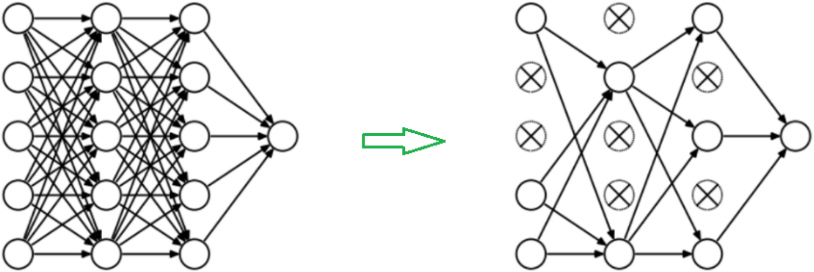
Simple change to the gradient:

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

Introduce *weight decaying effect*

Dropout

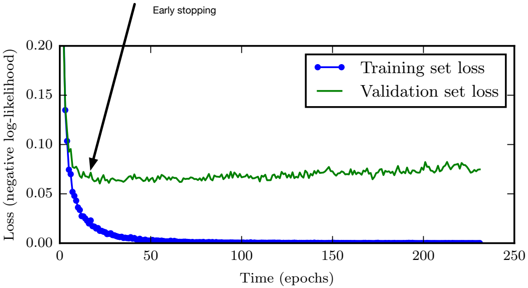
Independently delete each neuron with a fixed probability (say 0.5), during each iteration of Backprop (only for training, not for testing)



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

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

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)

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

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

Outline

Acknowledgements

Not much math, a lot of empirical intuitions

Acknowledgements

Not much math, a lot of empirical intuitions

The materials borrow heavily from the following sources:

- Stanford Course CS231n: `http://cs231n.stanford.edu/`
- Dr. Ian Goodfellow's lectures on deep learning: `http://deeplearningbook.org`

Both website provides tons of useful resources: notes, demos, videos, etc.

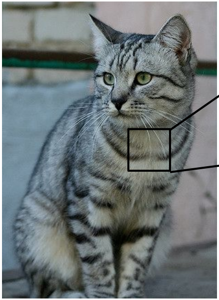
Image Classification: A core task in Computer Vision

This image by Nikita is licensed under CC-BY 2.0

(assume given set of discrete labels)
{dog, cat, truck, plane, ...}

→ cat

The Problem: Semantic Gap



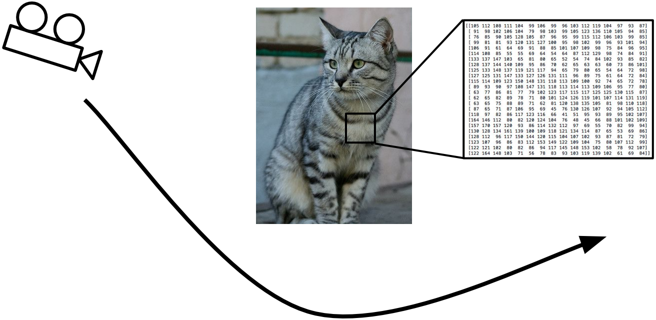

```
[[105 112 108 111 104  99 106  99  96 103 112 119 104  97  93  87]
 [ 91  98 102 106 104  79  98 103  99 105 123 136 110 105  94  85]
 [ 76  85  90 105 128 105  87  96  95  99 115 112 106 103  99  85]
 [ 99  81  81  93 120 131 127 100  95  98 102  99  96  93 101  94]
 [106  91  61  64  69  91  88  85 101 107 109  98  75  84  96  95]
 [114 108  85  55  55  69  64  54  64  87 112 129  98  74  84  91]
 [133 137 147 103  65  81  80  65  52  54  74  84 102  93  85  82]
 [128 137 144 140 109  95  86  70  62  65  63  63  60  73  86 101]
 [125 133 148 137 119 121 117  94  65  79  80  65  54  64  72  98]
 [127 125 131 147 133 127 126 131 111  96  89  75  61  64  72  84]
 [115 114 109 123 150 148 131 118 113 109 100  92  74  65  72  78]
 [ 89  93  90  97 108 147 131 118 113 114 113 109 106  95  77  80]
 [ 63  77  86  81  77  79 102 123 117 115 117 125 125 130 115  87]
 [ 62  65  82  89  78  71  80 101 124 126 119 101 107 114 131 119]
 [ 63  65  75  88  89  71  62  81 120 138 135 105  81  98 110 118]
 [ 87  65  71  87 106  95  69  45  76 130 126 107  92  94 105 112]
 [118  97  82  86 117 123 116  66  41  51  95  93  89  95 102 107]
 [164 146 112  80  82 120 124 104  76  48  45  66  88 101 102 109]
 [157 170 157 120  93  86 114 132 112  97  69  55  70  82  99  94]
 [130 128 134 161 139 100 109 118 121 134 114  87  65  53  69  86]
 [128 112  96 117 150 144 120 115 104 107 102  93  87  81  72  79]
 [123 107  96  86  83 112 153 149 122 109 104  75  80 107 112  99]
 [122 121 102  80  82  86  94 117 145 148 153 102  58  78  92 107]
 [122 164 148 103  71  56  78  83  93 103 119 139 102  61  69  84]]
```

What the computer sees

An image is just a big grid of numbers between [0, 255]:

e.g. 800 x 600 x 3
(3 channels RGB)

Challenges: Viewpoint variation



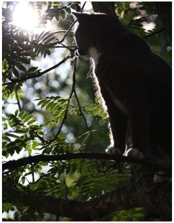
All pixels change when
the camera moves!

Challenges: Illumination

This image is CC0 1.0 public domain

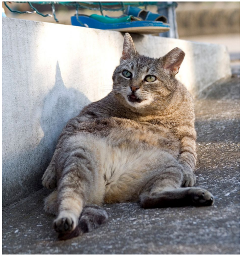
This image is CC0 1.0 public domain

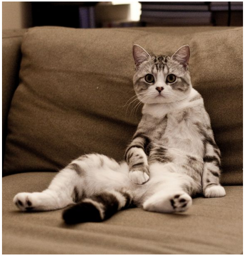
This image is CC0 1.0 public domain



This image is CC0 1.0 public domain

Challenges: Deformation





Challenges: Occlusion

This image is CC0 1.0 public domain

This image is CC0 1.0 public domain

This image by jonsson is licensed under CC-BY 2.0

Challenges: Background Clutter

Challenges: Intraclass variation

This image is CC0 1.0 public domain

Fundamental problems in vision

The key challenge

How to train a model that can tolerate all those variations?

Fundamental problems in vision

The key challenge

How to train a model that can tolerate all those variations?

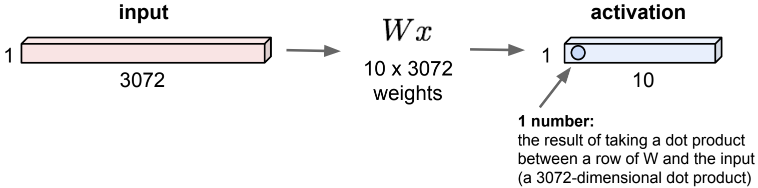
Main ideas

- need a lot of data that exhibits those variations
- need more specialized models to capture the invariance

Issues of standard NN for image inputs

Fully Connected Layer

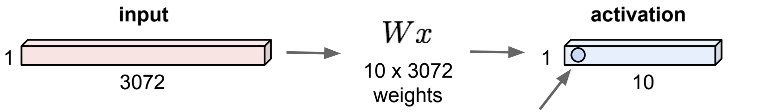
32x32x3 image -> stretch to 3072 x 1



Issues of standard NN for image inputs

Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



1 number:
the result of taking a dot product
between a row of W and the input
(a 3072-dimensional dot product)

Fei-Fei Li & Justin Johnson & Serena Yeung   Lecture 5 - 27   April 18, 2017

Spatial structure is lost!

Solution: Convolutional Neural Net (ConvNet/CNN)

A special case of fully connected neural nets

Solution: Convolutional Neural Net (ConvNet/CNN)

A special case of fully connected neural nets

- usually consist of **convolution layers**, ReLU layers, **pooling layers**, and regular fully connected layers

Solution: Convolutional Neural Net (ConvNet/CNN)

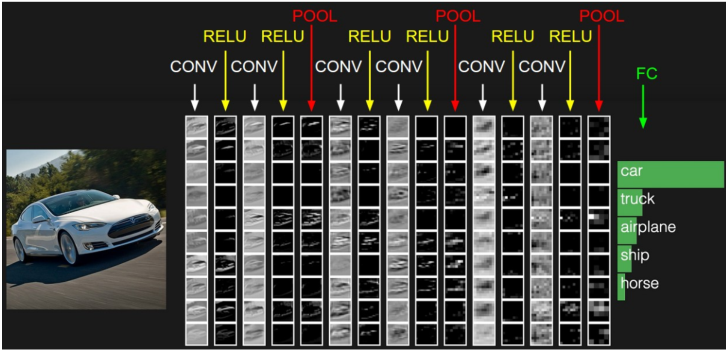
A special case of fully connected neural nets

- usually consist of **convolution layers**, ReLU layers, **pooling layers**, and regular fully connected layers
- key idea: *learning from low-level to high-level features*

Solution: Convolutional Neural Net (ConvNet/CNN)

A special case of fully connected neural nets

- usually consist of **convolution layers**, ReLU layers, **pooling layers**, and regular fully connected layers
- key idea: *learning from low-level to high-level features*

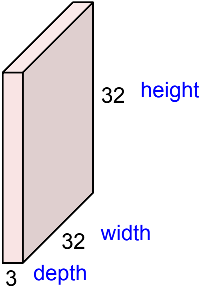


Convolution layer

Arrange neurons as a **3D volume** naturally

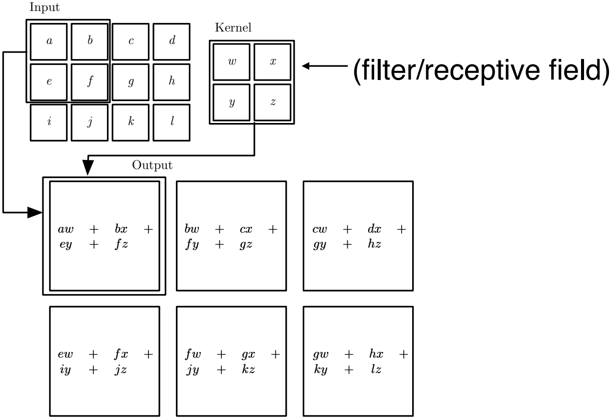
Convolution Layer

32x32x3 image -> preserve spatial structure



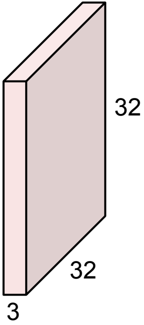
Convolution

2D Convolution

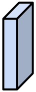


Convolution Layer

32x32x3 image



5x5x3 filter

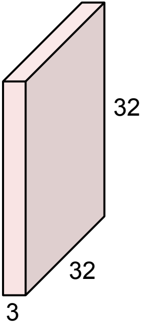


Convolve the filter with the image
i.e. “slide over the image spatially, computing dot products”

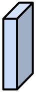
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

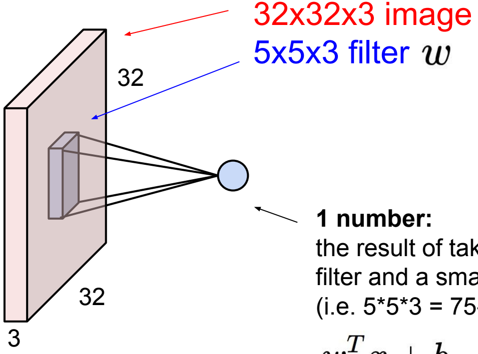


5x5x3 filter



Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

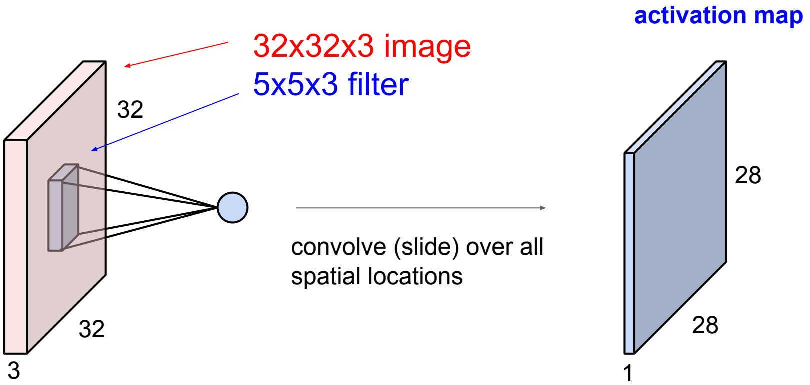
Convolution Layer



1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

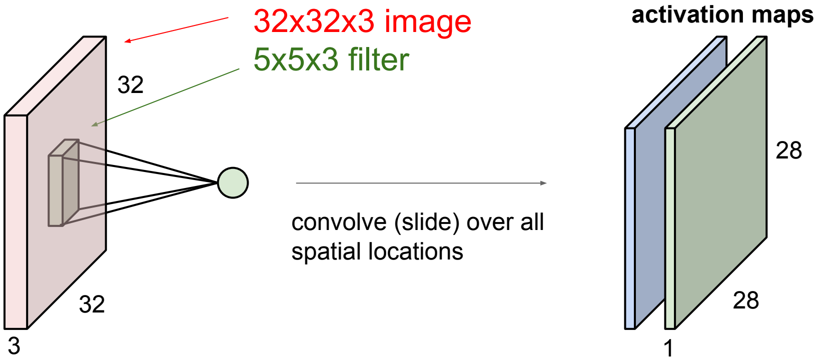
$$w^T x + b$$

Convolution Layer

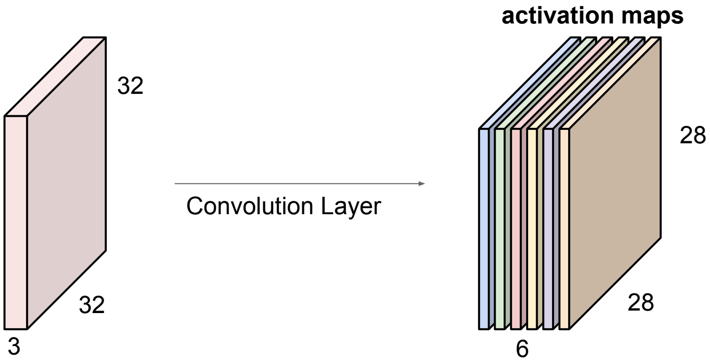


Convolution Layer

consider a second, green filter

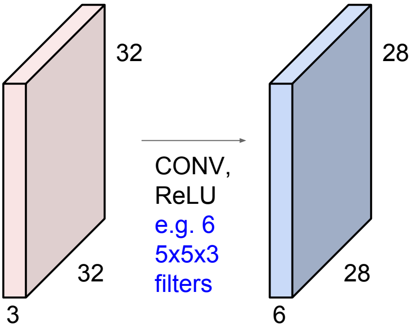


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

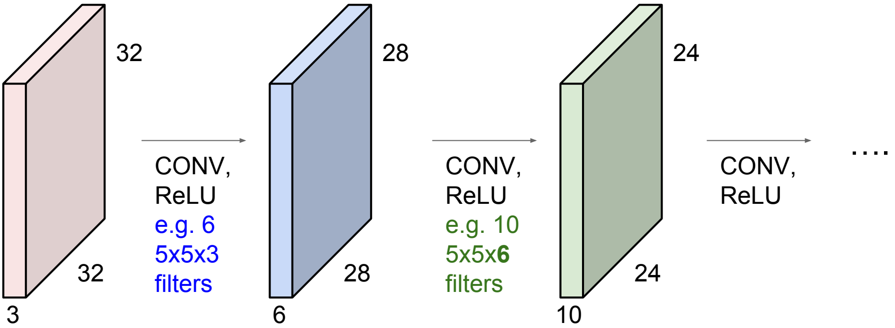


We stack these up to get a "new image" of size 28x28x6!

Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



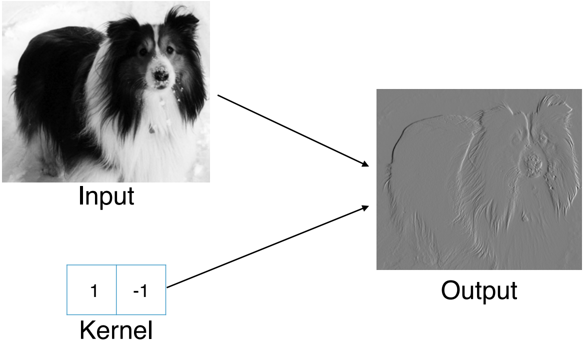
Why convolution makes sense?

Main idea: **if a filter is useful at one location, it should be useful at other locations.**

Why convolution makes sense?

Main idea: **if a filter is useful at one location, it should be useful at other locations.**

A simple example why filtering is useful



Connection to fully connected NNs

A convolution layer is a special case of a fully connected layer:

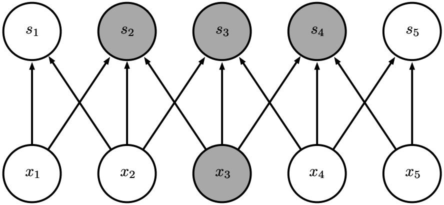
Connection to fully connected NNs

A convolution layer is a special case of a fully connected layer:

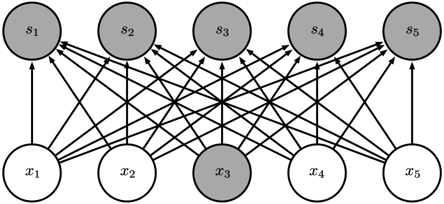
- filter = weights with **sparse connection**

Local Receptive Field Leads to Sparse Connectivity (affects less)

Sparse connections due to small convolution kernel



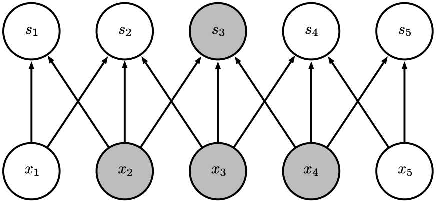
Dense connections



(Goodfellow 2016)

Sparse connectivity: being affected by less

Sparse connections due to small convolution kernel



Dense connections

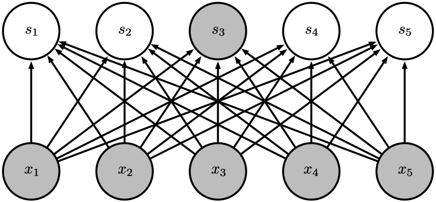


Figure 9.3

(Goodfellow 2016)

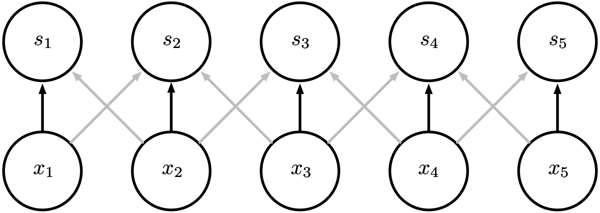
Connection to fully connected NNs

A convolution layer is a special case of a fully connected layer:

- filter = weights with **sparse connection**
- **parameters sharing**

Parameter Sharing

Convolution shares the same parameters across all spatial locations



Traditional matrix multiplication does not share any parameters

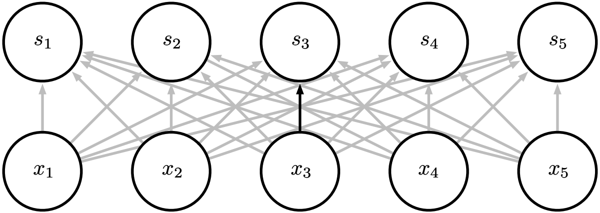


Figure 9.5

(Goodfellow 2016)

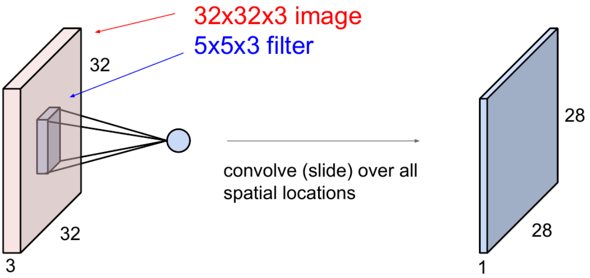
Connection to fully connected NNs

A convolution layer is a special case of a fully connected layer:

- filter = weights with **sparse connection**
- **parameters sharing**

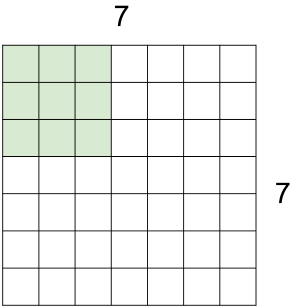
Much fewer parameters! Example (ignore bias terms):

- FC: $(32 \times 32 \times 3) \times (28 \times 28) \approx 2.4M$
- CNN: $5 \times 5 \times 3 = 75$



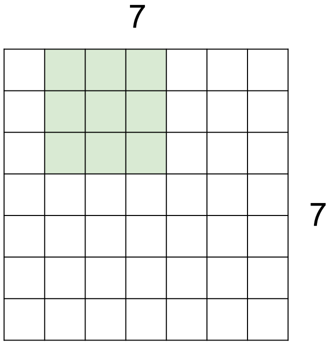
Spatial arrangement: stride and padding

A closer look at spatial dimensions:



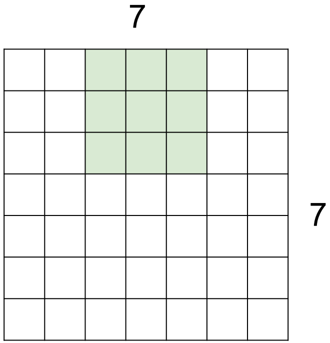
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



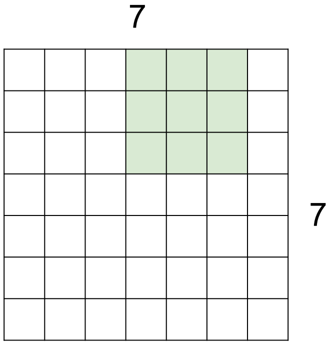
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



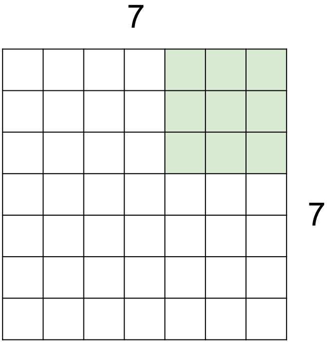
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

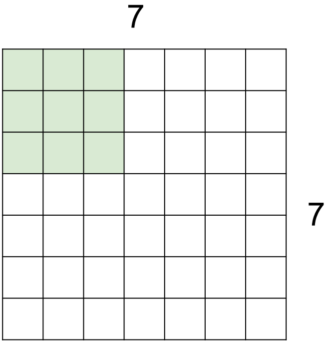
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

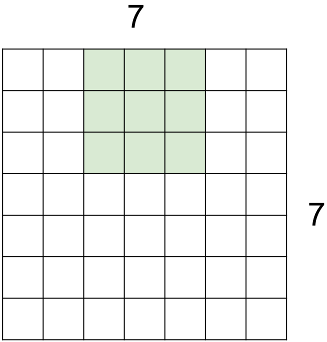
=> 5x5 output

A closer look at spatial dimensions:



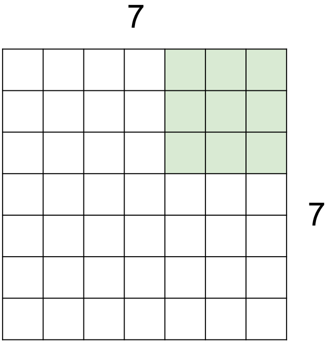
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



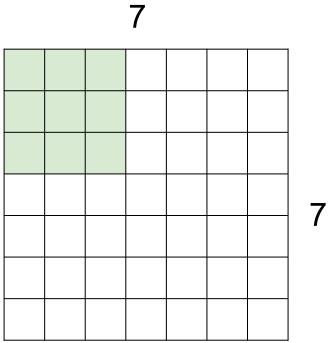
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



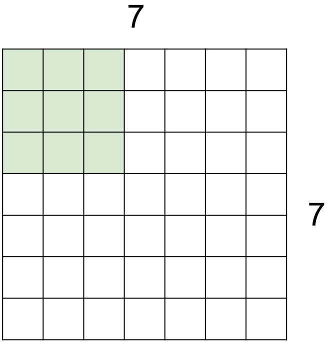
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



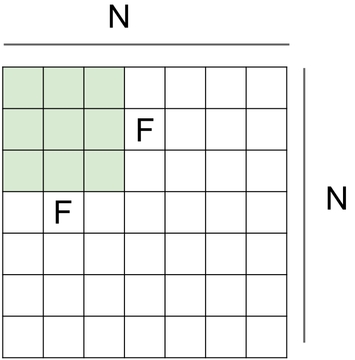
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

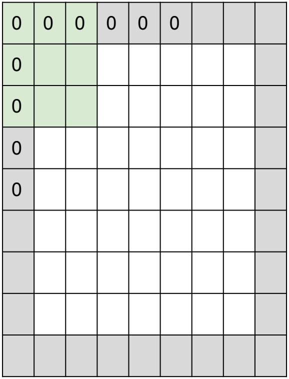
doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.



Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

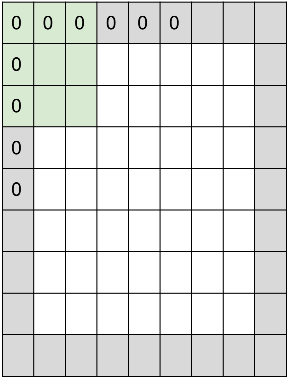
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

(recall:)
(N - F) / stride + 1

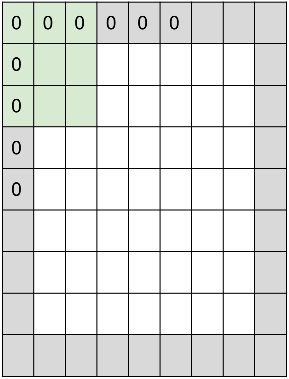
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

In practice: Common to zero pad the border

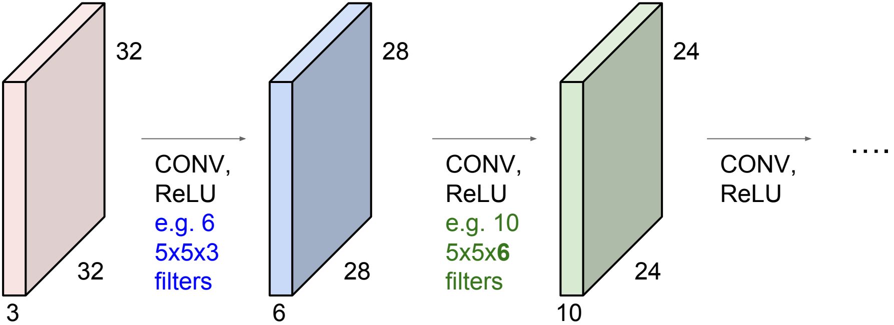


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Remember back to…

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.



Summary for convolution layer

Input: a volume of size $W_1 \times H_1 \times D_1$

Summary for convolution layer

Input: a volume of size $W_1 \times H_1 \times D_1$

Hyperparameters:

- K filters of size $F \times F$
- stride S
- amount of zero padding P (for one side)

Summary for convolution layer

Input: a volume of size $W_1 \times H_1 \times D_1$

Hyperparameters:

- K filters of size $F \times F$
- stride S
- amount of zero padding P (for one side)

Output: a volume of size $W_2 \times H_2 \times D_2$ where

- $W_2 =$
- $H_2 =$
- $D_2 =$

Summary for convolution layer

Input: a volume of size $W_1 \times H_1 \times D_1$

Hyperparameters:

- K filters of size $F \times F$
- stride S
- amount of zero padding P (for one side)

Output: a volume of size $W_2 \times H_2 \times D_2$ where

- $W_2 = (W_1 + 2P - F)/S + 1$
- $H_2 =$
- $D_2 =$

Summary for convolution layer

Input: a volume of size $W_1 \times H_1 \times D_1$

Hyperparameters:

- K filters of size $F \times F$
- stride S
- amount of zero padding P (for one side)

Output: a volume of size $W_2 \times H_2 \times D_2$ where

- $W_2 = (W_1 + 2P - F)/S + 1$
- $H_2 = (H_1 + 2P - F)/S + 1$
- $D_2 =$

Summary for convolution layer

Input: a volume of size $W_1 \times H_1 \times D_1$

Hyperparameters:

- K filters of size $F \times F$
- stride S
- amount of zero padding P (for one side)

Output: a volume of size $W_2 \times H_2 \times D_2$ where

- $W_2 = (W_1 + 2P - F)/S + 1$
- $H_2 = (H_1 + 2P - F)/S + 1$
- $D_2 = K$

Summary for convolution layer

Input: a volume of size $W_1 \times H_1 \times D_1$

Hyperparameters:

- K filters of size $F \times F$
- stride S
- amount of zero padding P (for one side)

Output: a volume of size $W_2 \times H_2 \times D_2$ where

- $W_2 = (W_1 + 2P - F)/S + 1$
- $H_2 = (H_1 + 2P - F)/S + 1$
- $D_2 = K$

#parameters: $(F \times F \times D_1 + 1) \times K$ weights

Summary for convolution layer

Input: a volume of size $W_1 \times H_1 \times D_1$

Hyperparameters:

- K filters of size $F \times F$
- stride S
- amount of zero padding P (for one side)

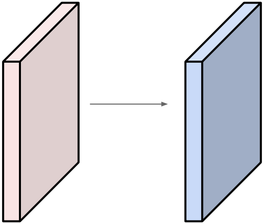
Output: a volume of size $W_2 \times H_2 \times D_2$ where

- $W_2 = (W_1 + 2P - F)/S + 1$
- $H_2 = (H_1 + 2P - F)/S + 1$
- $D_2 = K$

#parameters: $(F \times F \times D_1 + 1) \times K$ weights

Common setting: $F = 3, S = P = 1$

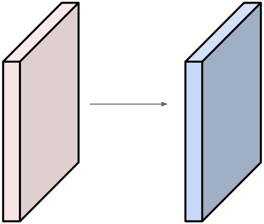
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?

Examples time:



Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

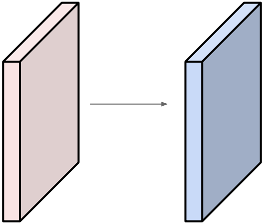
Output volume size:

(32+2*2-5)/1+1 = 32 spatially, so

32x32x10

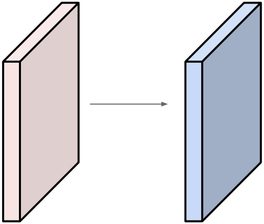
Examples time:

Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2



Number of parameters in this layer?

Examples time:



Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

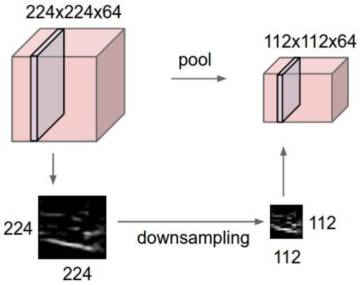
each filter has 5*5*3 + 1 = 76 params (+1 for bias)

=> 76*10 = **760**

Another element: pooling

Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:



Pooling

Similar to a filter, except

- depth is always 1

Pooling

Similar to a filter, except

- depth is always 1
- different operations: average, L2-norm, max

Pooling

Similar to a filter, except

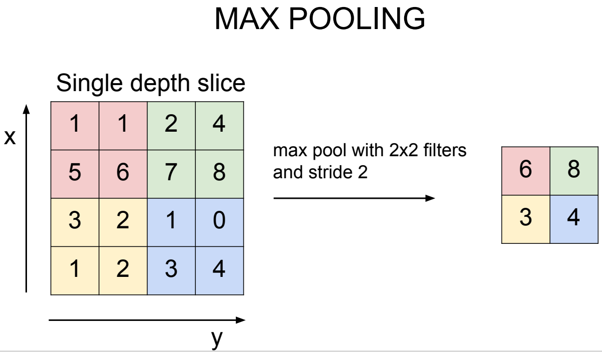
- depth is always 1
- different operations: average, L2-norm, max
- no parameters to be learned

Pooling

Similar to a filter, except

- depth is always 1
- different operations: average, L2-norm, max
- no parameters to be learned

Max pooling with 2×2 filter and stride 2 is very common



Putting everything together

Typical architecture for CNNs:

Input $\rightarrow$ [[Conv $\rightarrow$ ReLU]*N $\rightarrow$ Pool?]*M $\rightarrow$ [FC $\rightarrow$ ReLU]*Q $\rightarrow$ FC

Putting everything together

Typical architecture for CNNs:

Input $\rightarrow$ [[Conv $\rightarrow$ ReLU]*N $\rightarrow$ Pool?]*M $\rightarrow$ [FC $\rightarrow$ ReLU]*Q $\rightarrow$ FC

Common choices: $N \leq 5, Q \leq 2$, M is large

Putting everything together

Typical architecture for CNNs:

Input $\rightarrow$ [[Conv $\rightarrow$ ReLU]*N $\rightarrow$ Pool?]*M $\rightarrow$ [FC $\rightarrow$ ReLU]*Q $\rightarrow$ FC

Common choices: $N \leq 5, Q \leq 2$, M is large

Well-known CNNs: LeNet, AlexNet, ZF Net, GoogLeNet, VGGNet, etc.

All achieve excellent performance on image classification tasks.

How to train a CNN?

How do we learn the filters/weights?

How to train a CNN?

How do we learn the filters/weights?

Essentially the same as FC NNs: apply **SGD/backpropagation**