Statistical Learning VIII

1. Bagging + random forests
2. Boosting
3. Deep learning / neural networks
   a) neural networks
   b) size
   c) preventing overfitting
   d) (online) gradient descent and backpropagation

1. Bagging $T = \binom{Y_1}{X_1}, \ldots, \binom{Y_N}{X_N}$

Motivation: suppose we have high variance method (e.g. decision trees without pruning). How do we reduce variance?

Bootstrap + Aggregation

high variance = very different (e.g. trees) estimates when run on new data set.

- simulate many data sets by bootstrap and "average out" the variance.

Bootstrap:

- Z_b is sample of size N, sampled with replacement from T.

$b = 1, \ldots, B$

- Train estimators $\hat{f}_b$ for each Z_b

Aggregation:

for regression: $\hat{f}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}_b(x)$ is average

for classification:

* majority vote among $f_1, \ldots, f_B$
* if classifiers provide ~~weights~~ estimates $\hat{p}_1(Y=1|x), \ldots, \hat{p}_B(Y=1|x)$ then average those.

Bagging reduces variance, but does not help (or very little) if model cannot express ~~even~~ what we need to learn, i.e. has large bias. Figures 8.9, 8.10, 8.12

Bootstrap samples: want diversity of estimators $\hat{f}_b$ (e.g. ~~many~~ diverse set of trees)

Random forests: (optionally: see Ch. 15)

decision trees (without pruning) + bagging + some small extra tricks (e.g. to increase diversity among generated trees)

~~One of the best methods "out of the box", without engineering features or tuning parameters.~~

2. Boosting

"weak learners": high bias, low variance (assume that only slightly better than random guessing...)

(e.g. decision stumps = decision trees with only one binary decision)

How can be turn these into very good learners? (Fig 8.12, right)

$\begin{pmatrix} y_1 \\ x_1 \end{pmatrix}, \begin{pmatrix} y_2 \\ x_2 \end{pmatrix}, \ldots, \begin{pmatrix} y_N \\ x_N \end{pmatrix}$ $\quad y \in \{-1, +1\}$

$\omega_1 \quad \omega_2 \quad \omega_N$

Fig. 10.1, 10.2: to train G_m, increase weights for examples misclassified by a combination of previous classifiers.

Algorithm 10.1

- force it to concentrate on difficult examples)
- But why like AdaBoost?

Forward Stagewise Additive Modeling

Additive models: $f(x) = \sum_{m=1}^{M} \beta_m b_{\gamma_m}(x)$ (β_m: weights; b_{γ_m}: basis functions with parameters γ_m)

Fit by ERM:

$$\min_{(\beta_m, \gamma_m)_{m=1,\ldots,M}} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b_{\gamma_m}(x)\right)$$

often computationally difficult.

FSAM:

For $m = 1, \ldots, M$:

$$f_m(x) = f_{m-1}(x) + \beta_m b_{\gamma_m}(x)$$

Find parameters β_m, b_{γ_m} by ERM:

$$\min_{\beta_m, \gamma_m} \sum_{i=1}^{N} L\left(y_i, f_{m-1}(x) + \beta_m b_{\gamma_m}(x)\right)$$

Boosting as FSAM:

basis functions: G_m

exponential loss: $L(y, f(x)) = e^{-y \cdot f(x)}$

$$\beta_m, G_m = \underset{\beta, G}{\operatorname{argmin}} \sum_{i=1}^{N} e^{-y_i (f_{m-1}(x_i) + \beta \cdot G(x_i))}$$

$$= \underset{\beta, G}{\operatorname{argmin}} \sum_{i=1}^{N} w_i^{(m)} \cdot e^{-\beta y_i G(x_i)}$$

$$w_i^{(m)} = \frac{e^{-y_i f_{m-1}(x_i)}}{\text{normalisation}}$$

normalisation ↑ make weights sum to 1

equivalent to AdaBoost weights with alpha_m = 2 * beta_m

Solve first for G given any fixed β:

$$\min_G \sum_{i: y_i = G(x_i)} w_i^{(m)} e^{-\beta} + \sum_{i: y_i \neq G(x_i)} w_i^{(m)} e^{\beta}$$

$$= \min_G (e^{\beta} - e^{-\beta}) \sum_{i=1}^{N} w_i^{(m)} L_{01}(y_i, G(x_i)) + e^{-\beta} \underbrace{\sum_{i=1}^{N} w_i^{(m)}}_{1}$$

weighted ERM:

$$\min_G \sum_{i=1}^{N} w_i^{(m)} L_{01}(y_i, G(x_i)) \leftarrow \text{err}_m$$

(view 2a in AdaBoost as approximately solving this)

Solve for β:

$$0 = \frac{d}{d\beta} (e^{\beta} - e^{-\beta}) \text{err}_m + e^{-\beta} = (e^{\beta} + e^{-\beta}) \text{err}_m - e^{-\beta}$$

$$(e^{2\beta} + 1) \text{err}_m = 1$$

$$e^{2\beta} \text{err}_m = 1 - \text{err}_m$$

$$e^{2\beta} = \frac{1 - \text{err}_m}{\text{err}_m}$$

$$\beta = \frac{1}{2} \log \frac{1 - \text{err}_m}{\text{err}_m}$$

is what boosting does!

Compare Fig. 10.3 and 10.2

training error to 0

but exponential loss and test set error keep decreasing!

Why?

Why exponential loss?

Estimate in population:

$$f_B(x) = \underset{f(x)}{\operatorname{argmin}} \; \mathbb{E}_{y|x}\left[e^{-y \cdot f(x)}\right]$$

$$= \frac{1}{2} \log \frac{\Pr(y=1|x)}{\Pr(y=-1|x)}$$

so exponential loss tries to estimate log odds of classes.

Slide: surrogate losses

* squared error bad because punishes large positive margin
* exponential loss: really dislikes large negative margin

⇓

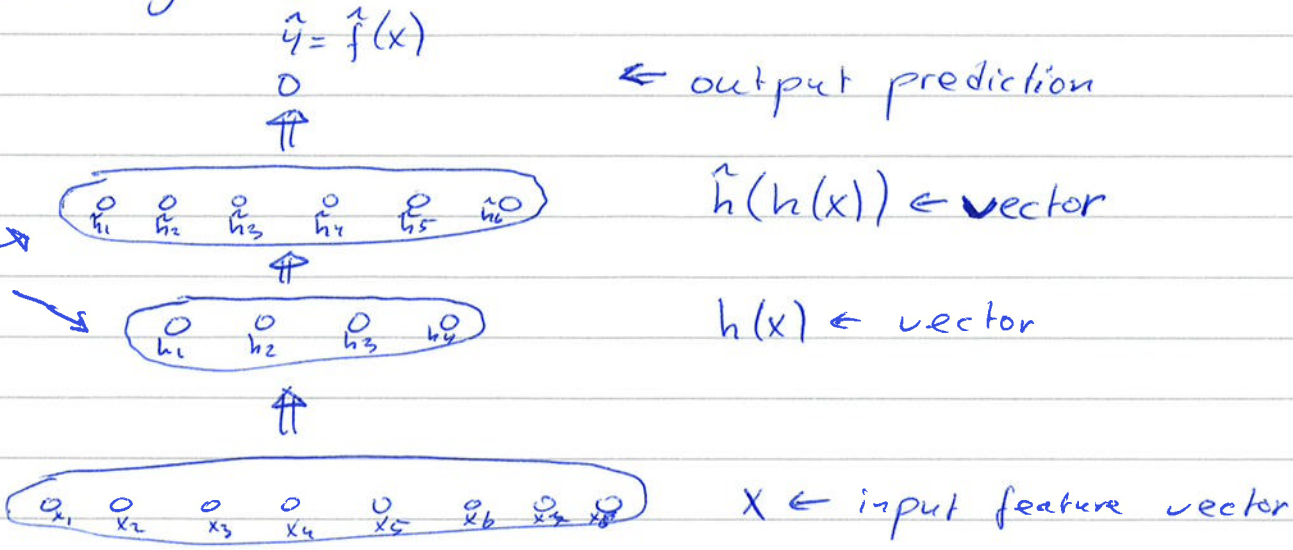
AdaBoost gives large weights to observations that are misclassified with large margin

⇓

not robust in settings where this is unavoidable: large noise or mislabeled data

Then might be better to use alternative loss like binomial deviance, although might be computationally more challenging.

Deep learning = Big neural networks



Learn feature transformations from the data!

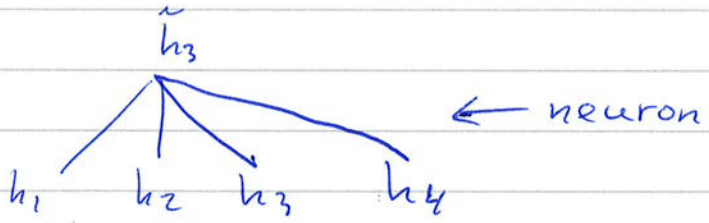
└ completely replaced manually designed SWIFT features in computer vision ← took 10 years to design these

* Can have >3 layers
* Intermediate vectors often very large

} "deep" learning

* Loss $L(y, \hat{y})$ can be ~~easy~~ anything. E.g. quadratic for regression, logistic for classification, etc

How do we learn transformations between layers? Zoom in.



$$\tilde{a}_3 = \sum_{i=1}^{4} w_{h_i \to \tilde{h}_3} \cdot h_i + \text{intercept}_{\tilde{h}_3}$$

linear "activation"

parameters

$$\tilde{h}_3 = \sigma(\tilde{a}_3)$$

σ ← activation function (nonlinear)

- sigmoid: $\sigma(a) = \frac{1}{1+e^{-a}}$

- tanh: $\sigma(a) = \frac{1-e^{-2a}}{1+e^{-2a}} = 2 \cdot \text{sigmoid}(2a) - 1$

- rectified linear unit: $\sigma(a) = \max\{a, 0\}$ ← (helped a lot in performance)
  (ReLu) ← recently popular
  piece-wise linear

- ...

Size:
- 2 or 3 hidden layers often enough
- every neuron has its own parameters
- modern network may have
  inputs: 65536 = 256 x 256 color pixels
  layers: 500
  500
  2000

rough estimate nr of parameters:
$65536 \times 2000 + 2000 \times 500 + 500 \times 500 + 500 \times 1$
$\approx$ 130 million parameters

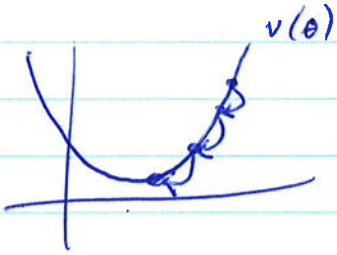
So will overfit like crazy...

How to prevent overfitting:
1. Very large training sets
2. Exploit domain knowledge to reduce nr of params: images have locally similar structures, so tie parameters in first layer together (convolutional neural networks)
3. Regularize (L_1, L_2)
4. Do not find ERM parameters (which is impossible anyway, because it is a non-convex function of the parameters)

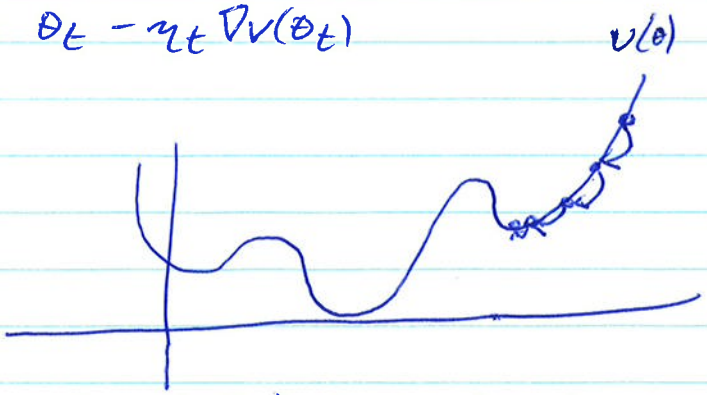
How to estimate parameters?

θ is vector with all weights (e.g. 150 million parameters)

Gradient descent: $\theta_{t+1} = \theta_t - \eta_t \nabla v(\theta_t)$



works for convex $v(\theta)$



(hopefully) finds local minimum for non-convex $v(\theta)$

Solution: accept we find local minimum

Big open question: why does it find good local minima in practice? ← appears to be better with larger networks.

Actually use stochastic GD on $v(\theta) = \sum_{i=1}^{N} L(Y_i, f_\theta(x_i))$

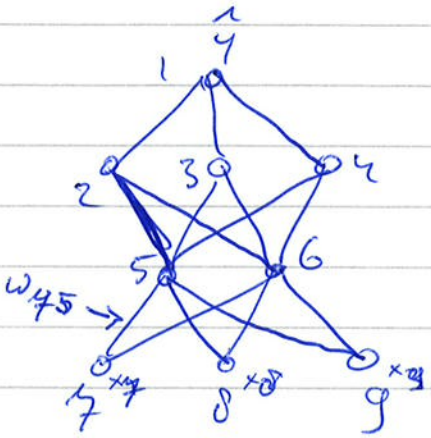
Need to compute gradients $\nabla L(Y_i, f_\theta(x_i))$ fast!

How to compute gradients $\nabla_\theta L(y_t, \hat{f}_\theta(x_t))$?

130 million parameters: only have time for fixed nr. of computations per parameter

Solution: Backpropagation ← use chain rule for derivatives many times:

$$\frac{\delta a}{\delta b} = \frac{\delta a}{\delta c} \cdot \frac{\delta c}{\delta b}$$



For each neuron in o:

$o_m = \sigma(a_m)$ (o: output, a: activation) $\quad m = 2, \dots, 9$

$o_1 = \hat{y}$

1. $\frac{\delta L(y_t, \hat{f}_\theta(x_t))}{\delta o_1}$ is $\frac{\delta L(y_t, \hat{y})}{\delta \hat{y}}$ is derivative of the loss

2. chain rule:

Suppose we have $\frac{\delta L(y_t, \hat{f}_\theta(x_t))}{\delta o_5}$, then

$$\frac{\delta L(y_t, \hat{f}_\theta(x_t))}{\delta w_{45}} = \frac{\delta L(y_t, \hat{f}_\theta(x_t))}{\delta o_5} \cdot \frac{\delta o_5}{\delta w_{45}}$$

$\uparrow$ 9

$$\frac{\delta o_5}{\delta w_{45}} = \frac{\delta o_5}{\delta a_5} \cdot \frac{\delta a_5}{\delta w_{45}}$$

$$\frac{\delta o_5}{\delta a_5} = \frac{\delta \sigma(a_5)}{\delta a_5} = \sigma'(a_5)$$

$$\frac{\delta a_5}{\delta w_{45}} = \frac{\delta \sum_{i=4}^{9} w_{i5} \cdot x_i}{\delta w_{45}} = x_4$$

3. Work backwards from output to inputs to compute partial derivatives w.r.t. each o_m if we already have these partial derivatives higher up in the network.

already computed

vector chain rule:

$$\frac{\delta L(y, f_\theta(x))}{\delta o_5} = \frac{\delta L(y, f_\theta(x))}{\delta o_2} \cdot \frac{\delta o_2}{\delta o_5} + \frac{\delta L(y, f_\theta(x))}{\delta o_3} \cdot \frac{\delta o_3}{\delta o_5} + \frac{\delta L(y, f_\theta(x))}{\delta o_4} \cdot \frac{\delta o_4}{\delta o_5}$$

$$\frac{\delta o_2}{\delta o_5} = \frac{\delta o_2}{\delta a_2} \cdot \frac{\delta a_2}{\delta o_5}$$

$\sigma'(a_2)$

$a_2 = w_{52} \cdot o_5 + w_{62} \cdot o_6$

$$\frac{\delta(a_2)}{\delta o_5} = w_{52}$$

Conclusion: only need to do fixed (nr of) operations for each element of the network.

↳ compute gradient vector quickly