M2177.003100
Deep Learning

[5: Regularization]

Electrical and Computer Engineering
Seoul National University

(last compiled at 12:19:00 on 2020/09/20)

Outline

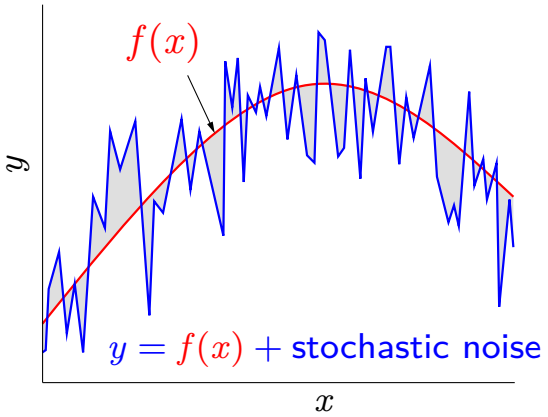
References

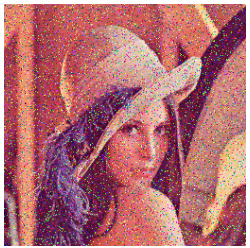
- *Deep Learning* by Goodfellow, Bengio and Courville Link
  - Chapter 7
- online resources:
  - *Deep Learning Specialization (coursera)* Link
  - *Stanford CS231n: CNN for Visual Recognition* Link

Outline

Noise: part of y we cannot model

- stochastic/deterministic noise hurts learning by leading to overfitting





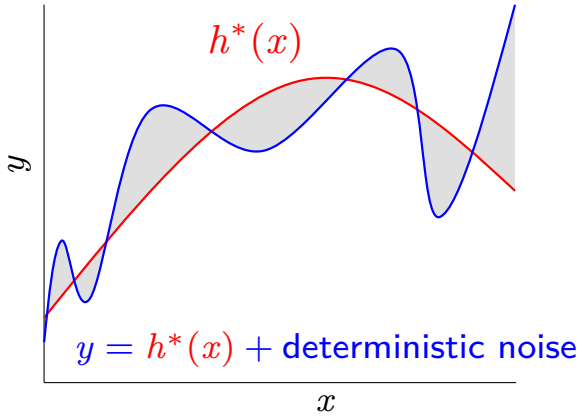


image sources: Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from Data*. AMLBook, 2012; Wilscy and Nair, "Fuzzy Approach for Restoring Color Images Corrupted with Additive Noise," in *Proceedings of the World Congress on Engineering*, 2008; `https://images.app.goo.gl/suQr73WuwNC2snkE7`

- human: good at extracting simple patterns
  - ignoring noise and complications
- computers: pay equal attention to all pixels
  - need help for Simplification
  - *e.g.* feature extraction, regularization, attention

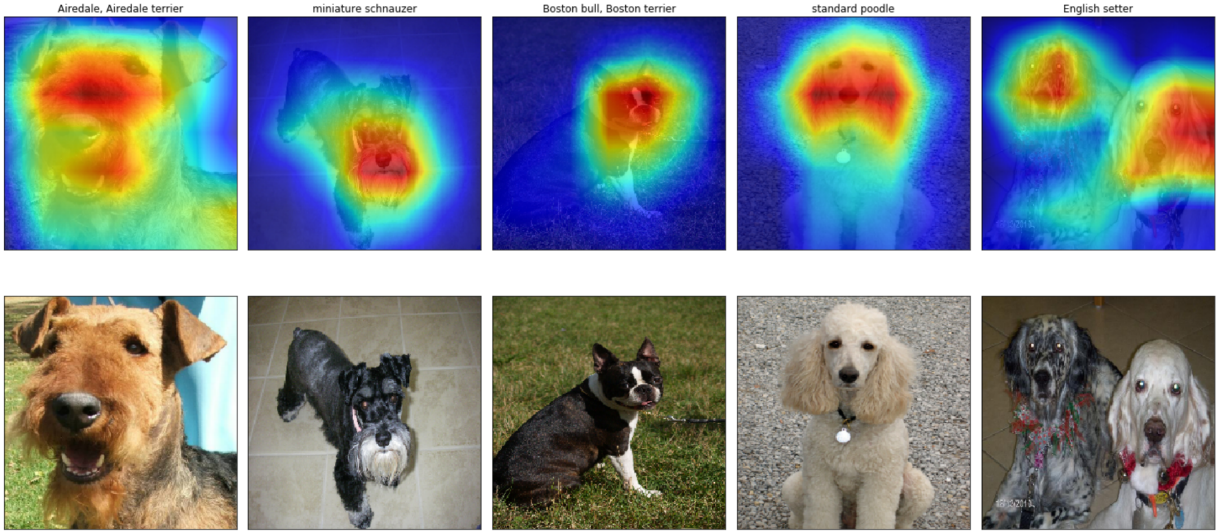


image source:
https://alexisbcook.github.io/2017/global-average-pooling-layers-for-object-localization/

Overfitting

- literally: fitting the data more than is warranted
- consequences
  - fitting observation (small $\mathrm{E}_{\mathrm{train}}$) no longer indicates decent $\mathrm{E}_{\mathrm{test}}$

  $\Rightarrow$ E_{train} alone is no longer good guide for learning
- observed when
  - a model is more complex than is necessary to represent target
  - the model uses its additional DOF to fit noise in data

  $\Rightarrow$ inferior final model
- ability to deal with overfitting
  - what separates professionals from amateurs

Regularization

- what is it?
  - a cure for tendency to fit noise, hence improving $\mathrm{E}_{\mathrm{test}}$
  - effective especially when noise is present
- how does it work?
  - constrain a model so that it can avoid fitting noise
- any side effects?
  - we may become incapable of fitting f(signal) faithfully

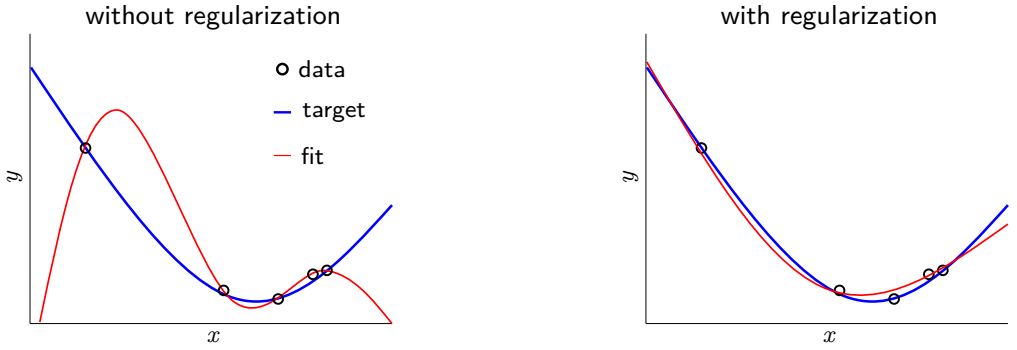


image source: Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from Data*. AMLBook, 2012

Regularization strategies

1. put extra constraints on a model

   e.g. add restrictions on parameter values

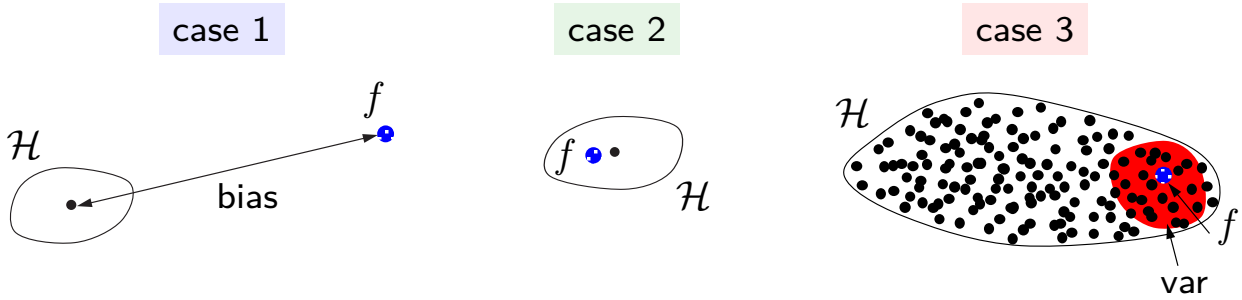
2. add extra terms in objective function
   - can be viewed as a soft constraint on parameter values

3. combine multiple hypotheses that explain training data
   - called ensemble methods

- extra constraints and penalties encode:
  - specific kinds of prior knowledge, or
  - generic preference for a simpler model to promote generalization

Goal of regularization

- three situations during training: the model either



case	model	main error	phenomenon
1	exclude f (true data generating process)	bias	underfitting
2	match f		
3	include f but also many others	variance	overfitting

- goal of regularization:
  - to take a model from regime 3 $\rightarrow$ regime 2

image source: Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from Data*. AMLBook, 2012

In the context of deep learning

- deep learning:
  - applied to extremely complicated domains (images/audio/text)

  $\Rightarrow$ true generation process involves simulating the entire universe

  $\Rightarrow$ true data generating process: almost certainly outside the model family

- this means: controlling the complexity of the model is
  - not a simple matter of finding the model of right size, # of parameters

- instead (almost always in practical deep learning)
  - best[1] model

  $=$ a large model that has been regularized appropriately

[1] in the sense of minimizing generalization error

Outline

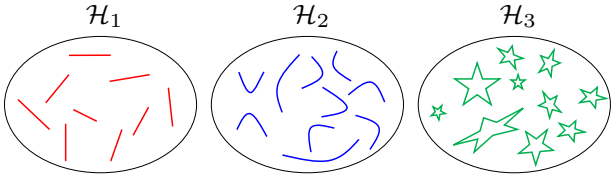
Regularization

- a process of introducing additional information, in order to
  - solve an ill-posed problem[2] or
  - prevent overfitting
- additional information: usually a penalty for complexity, *e.g.*,
  - restrictions for smoothness
  - bounds on the vector space norm
- two approaches
  1. mathematical: function approximation for ill-posed problems
  2. heuristic: handicapping the minimization of $\mathrm{E}_{\mathrm{train}}$
- regularization is as much an art as it is a science

[2] a well-posed problem (defined by Jacques Hadamard): (i) a solution exists (ii) the solution is unique (iii) the solution's behavior changes continuously with the initial conditions

VC generalization bound revisited

- f : unknown target function (objective of learning)
  - g : our (best) model learned from data (one of $h \in \mathcal{H}$)
  - $\mathcal{H}$: hypothesis set from which we choose g



- recall VC generalization bound (d_{VC} determined by which $\mathcal{H}$ is chosen):

$$\mathbb{P}[\ \overbrace{|\mathrm{E}_{\mathrm{train}}(f) - \mathrm{E}_{\mathrm{test}}(f)|}^{\text{generalization gap}} > \epsilon\ \underbrace{]}_{\text{bad event}} \leq \underbrace{4 \cdot (2N)^{\overbrace{d_{\mathrm{VC}}}^{\text{capacity}}} \cdot e^{-\frac{1}{8}\epsilon^2 N}}_{\text{VC bound}}$$

- from this we can derive: $\mathrm{E}_{\mathrm{test}}(h) \leq \mathrm{E}_{\mathrm{train}}(h) + \Omega(\mathcal{H})$ for all $h \in \mathcal{H}$

  i.e. $\mathrm{E}_{\mathrm{test}}$: bounded by penalty $\Omega(\mathcal{H})$ on model complexity

Core concept

- VC bound: $\mathrm{E}_{\text{test}}(h) \leq \mathrm{E}_{\text{train}}(h) + \Omega(\mathcal{H})$ for all $h \in \mathcal{H}$
  - good: we can fit data using a simple $\mathcal{H}$
- regularization: takes one step further
  - even better: we can fit using a simple $h \in \mathcal{H}$
- essence of regularization[3]
  - concoct $\Omega(h)$ for individual hypothesis
  - minimize combination of $\mathrm{E}_{\text{train}}(h)$ and $\Omega(h)$ (not $\mathrm{E}_{\text{train}}(h)$ alone)

  e.g. "weight decay" regularizer: minimize $\mathrm{E}_{\text{train}}(\boldsymbol{w}) + \frac{\lambda}{N}\boldsymbol{w}^\top\boldsymbol{w}$
  - $\Omega(h)$: now part of optimization objective
    - we can thus avoid overfitting by constraining learning process
- a member of even a large model family can be appropriately regularized

[3] we use $\Omega(h) \leftrightarrow \Omega(\boldsymbol{w})$ and $E(h) \leftrightarrow E(\boldsymbol{w})$ interchangeably

Augmented error

- combination of $\mathrm{E}_{\text{train}}$ and Ω (penalty on model complexity)

$$\mathrm{E}_{\text{aug}}(\boldsymbol{w}) = \mathrm{E}_{\text{train}}(\boldsymbol{w}) + \underbrace{\overbrace{\frac{\lambda}{N}}^{\text{regularization parameter}} \overbrace{\Omega(\boldsymbol{w})}^{\text{regularizer}}}_{\text{penalty term}} \tag{1}$$

  - better proxy for E_{test} than $\mathrm{E}_{\text{train}}$ (see Appendix for additional theory)

- example: weight decay regularizer (more on this soon):

$$\mathrm{E}_{\text{aug}}(\boldsymbol{w}) = \mathrm{E}_{\text{train}}(\boldsymbol{w}) + \frac{\lambda}{N} \boldsymbol{w}^\top \boldsymbol{w}$$

- need for regularization goes down as $N \uparrow$
  - we thus factor out $\frac{1}{N}$ — N↑ → VC bound ↓ → regularization 덜 중요해짐.
  - this allows optimal λ to be less sensitive to N

How to select regularizer Ω and parameter λ

- regularizer Ω "sparse" → L_1 regularizer ← heuristic
  - typically fixed ahead of time, before seeing data
  - sometimes problem itself dictates an appropriate regularizer
- choice of optimal λ ← principled
  - typically depends on data
  - overdose leads to underfitting ⇒ validation will tell us (see Appendix)

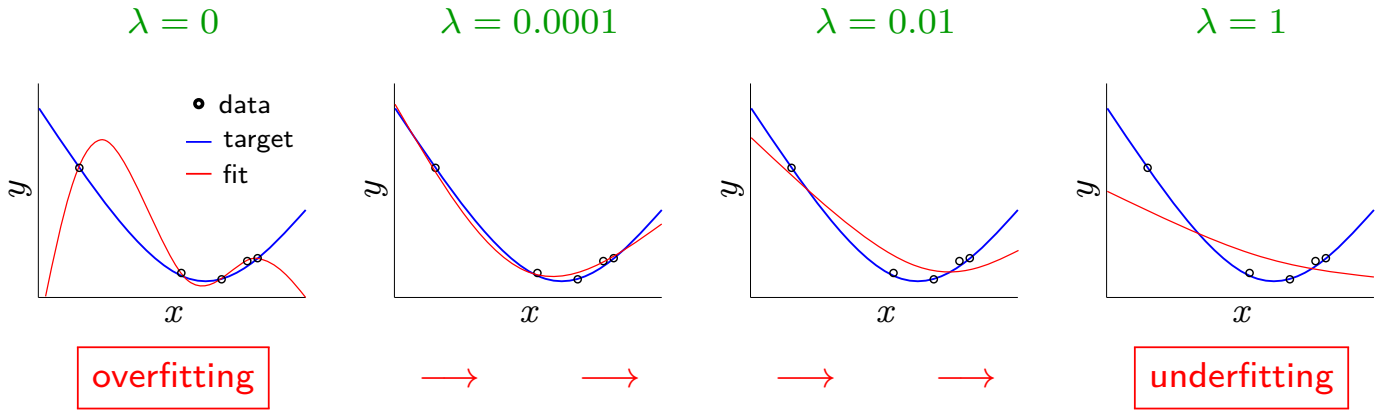


image source: Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from Data*. AMLBook, 2012

Considerations in neural nets

parameter	**weight**	**bias**
role	specifies how two variables interact	controls only a single variable
accurate fitting needs	more data (to observe both variables in various conditions)	less data than weights

- unregularized bias $\rightarrow$ not much variance
- regularizing bias $\rightarrow$ can give significant underfitting

- we thus use penalty Ω that leaves biases unregularized
  - penalize *only weights* of affine transformation at each layer

- convention in textbook:
  - $\boldsymbol{w}$ all the weights that should be affected by Ω
  - $\boldsymbol{\theta}$ all the parameters including both $\boldsymbol{w}$ and unregularized parameters

Outline

Most famous: l_1 and l_2 regularizers

- l_1 regularizer: aka lasso (statistics), *basis pursuit* (signal processing)
  - convex but not differentiable everywhere
  - variable shrinkage + selection: $\Rightarrow$ sparse solution

$$\Omega(\mathbf{w}) = ||\mathbf{w}||_1 = \sum_q |w_q|$$

- l_2 regularizer: aka ridge (statistics), *weight decay* (neural nets)
  - math friendly (convex/differentiable)
  - variable shrinkage only: shrinks w's of correlated x's

$$\Omega(\mathbf{w}) = ||\mathbf{w}||_2 = \mathbf{w}^\top \mathbf{w} = \sum_q w_q^2$$

- see Appendix for additional comparison

- why "weight decay"?

$$\begin{aligned} \boldsymbol{w} &\leftarrow \boldsymbol{w} - \epsilon \nabla \mathrm{E}_{\mathrm{aug}}(\boldsymbol{w}) \\ &= \boldsymbol{w} - \epsilon \nabla \mathrm{E}_{\mathrm{train}}(\boldsymbol{w}) - 2\epsilon \frac{\lambda}{N} \boldsymbol{w} \\ &= \underbrace{\left(1 - 2\epsilon \frac{\lambda}{N}\right)}_{\text{decay}} \boldsymbol{w} - \epsilon \nabla \mathrm{E}_{\mathrm{train}}(\boldsymbol{w}) \end{aligned}$$

- l_2 (weight decay) regularized cost function in neural nets:

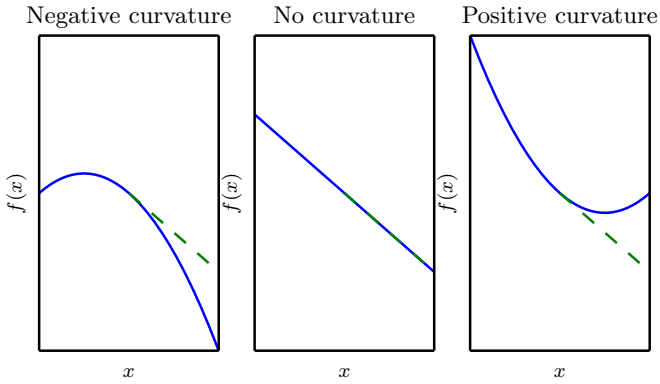
$$\underbrace{J(\boldsymbol{W}^{[1]}, \boldsymbol{b}^{[1]}, \ldots, \boldsymbol{W}^{[L]}, \boldsymbol{b}^{[L]})}_{\mathrm{E}_{\mathrm{aug}}(\boldsymbol{W}, \boldsymbol{b})} = \underbrace{\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)})}_{\mathrm{E}_{\mathrm{train}}(\boldsymbol{W}, \boldsymbol{b})} + \underbrace{\frac{\lambda}{m} \sum_{l=1}^{L} ||\boldsymbol{W}^{[l]}||_F^2}_{\Omega(\boldsymbol{W})\text{: regularizer}}$$

  - no regularization for bias
  - Frobenius norm of a matrix $\Leftrightarrow$ l_2 norm of a vector

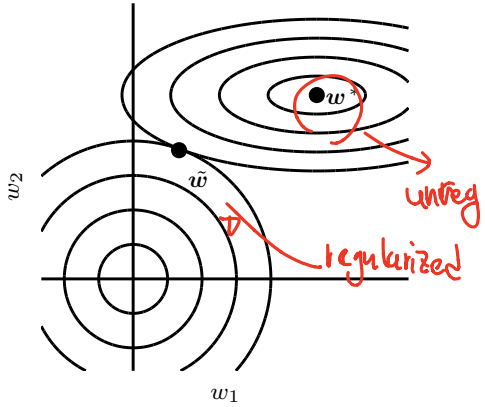
$$||A||_F^2 = \sum_{i,j} A_{i,j}^2$$

Effect of weight decay

- Hessian $\boldsymbol{H}$ of J gives curvature
  - the higher $\text{eigenval}(\boldsymbol{H})$ ↑
  - ⇒ the more curvature ↑



- rescale $\boldsymbol{w}^*$ along the axes defined by the eigenvectors of $\boldsymbol{H}$
  - scale factor for i-th eigenvector: $\frac{\text{eigenval}(\boldsymbol{H})_i}{\text{eigenval}(\boldsymbol{H})_i + \lambda}$



- w_1 direction (↔): $\text{eigenval}(\boldsymbol{H})_1$ small
  - J not increase much when moving away from $\boldsymbol{w}^*$ ⇒ J has no strong preference
  - regularizer has a Strong effect
- w_2 direction (↕): $\text{eigenval}(\boldsymbol{H})_2$ large
  - regularizer affects this direction little

image source: I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016

Other (generalized) regularizers

- Tikhonov regularizer
  - generalization of weight decay

$$\Omega(\mathbf{w}) = \mathbf{w}^\top \Gamma^\top \Gamma \mathbf{w} = \sum_p \sum_q w_p w_q \gamma_p \gamma_q$$

- elastic-net penalty
  - compromise between l_1 lasso ($\alpha = 1$) and l_2 ridge ($\alpha = 0$)

$$\Omega(\mathbf{w}) = \sum_q \left\{ \alpha |w_q| + \frac{1}{2}(1-\alpha) w_q^2 \right\}$$

Comparison

- regularization path
  - l_1 (lasso)
  - elastic net
  - l_2 (ridge)
- degree of freedom:

$$\sim C \sim \frac{1}{\lambda} \sim -\log \lambda$$

budget

  - df $\to 0$ as $\lambda \to \infty,\ C \to 0$
  - see Appendix for concept of C

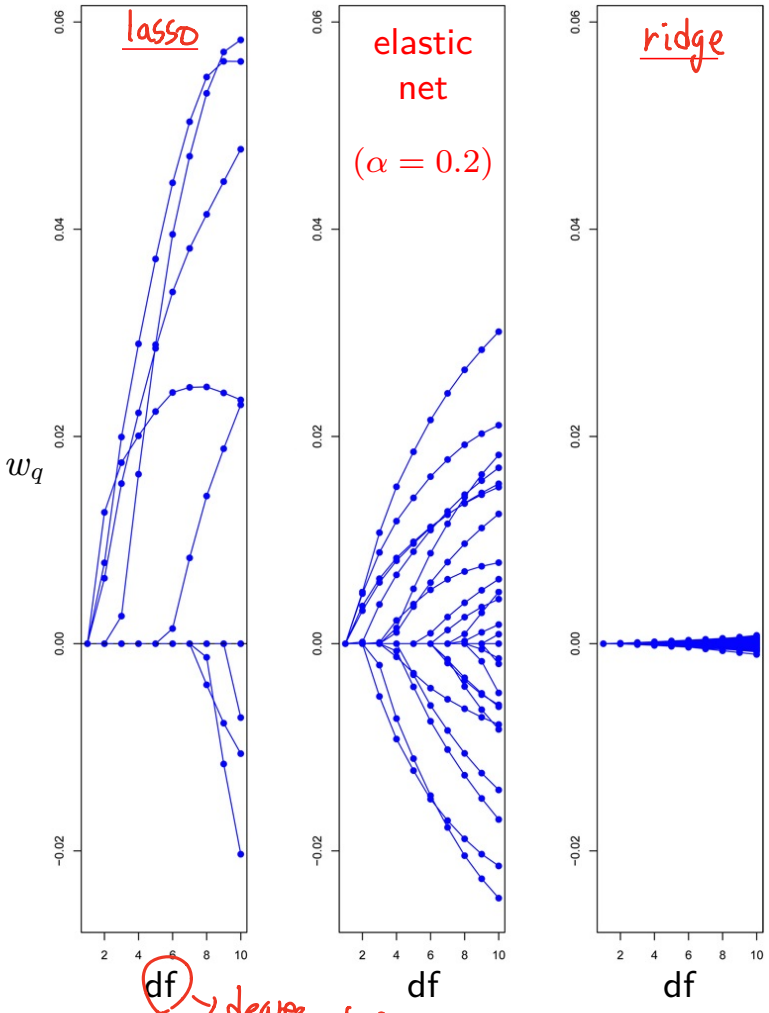
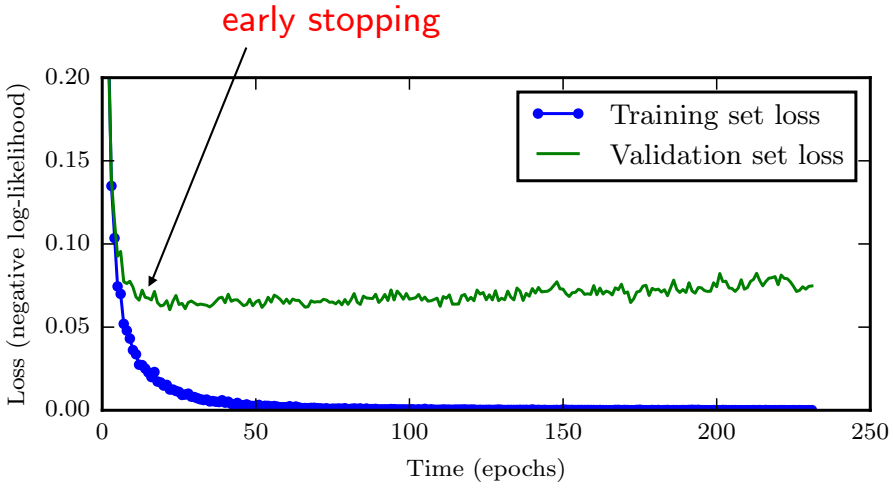


image source: Friedman, J., Hastie, T. and Tibshirani, R., 2010. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1), p.1.

Outline

Early stopping

- overfitting: $\mathrm{E}_{\mathrm{train}} \searrow$ but $\mathrm{E}_{\mathrm{dev}} \nearrow$
  - natural idea: how about stopping when E_{dev} gets low?
- early stopping: keep track of both $\mathrm{E}_{\mathrm{train}}$ and $\mathrm{E}_{\mathrm{dev}}$
  - stop training with lowest $\mathrm{E}_{\mathrm{dev}} \Rightarrow$ potentially better $\mathrm{E}_{\mathrm{test}}$



  - effective/simple $\rightarrow$ very popular also in deep learning

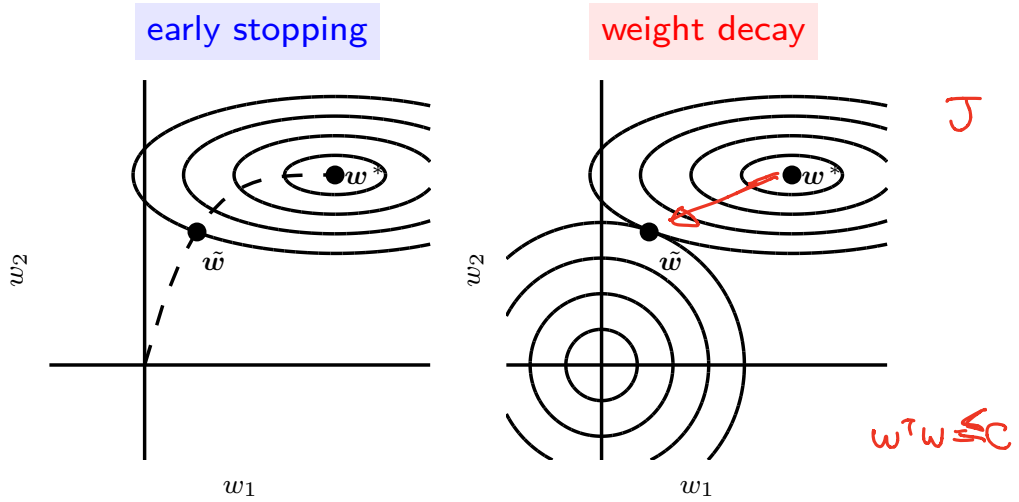
image source: I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016

- how it works (algorithm 7.1):
  - every time the error on validation set improves
    - store a copy of parameters ($\leftarrow$ returned when training terminates)
  - algorithm terminates when no improvement in $\mathrm{E}_{\mathrm{dev}}$

- advantages:
  1. very unobstrusive (unnoticeable) [c.f. weight decay]
     $\Rightarrow$ no change in learning dynamics (*e.g.* no need for adding Ω to J, etc)
  2. early stop $\Rightarrow$ fewer epochs $\Rightarrow$ computational savings
  3. leave extra data for additional training (algorithms 7.2 & 7.3)

- disadvantages:
  - must compute $\mathrm{E}_{\mathrm{dev}}$ periodically during training
    - mitigated by separate GPUs, small dev set, infrequent validation
  - additional memory to store best parameters (but negligible)

Comparison



- early stopping: more than mere restriction of trajectory length
  - monitors $\mathrm{E}_{\mathrm{dev}}$ to stop trajectory

  $\Rightarrow$ auto-determines correct amount of regularization (a big plus)

- weight decay
  - requires many training experiments with different hyperparameters

image source: I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016

Outline

Ensemble learning

- idea: make a strong model
  - by combine weak models
  - "model averaging"
- strong?
  - better bias/variance/accuracy...
- assumption
  - different models → different mistakes
  - averaging noise → zero
- most famous
  - (weighted) voting
  - bagging
  - boosting

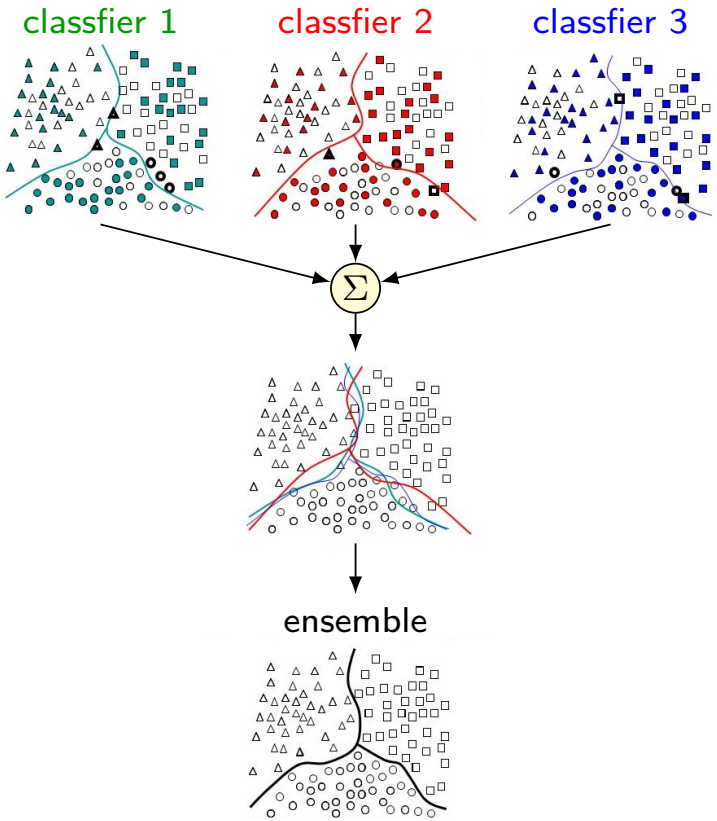


image source: http://www.scholarpedia.org/article/Ensemble_learning

Bagging (bootstrap aggregating)

- combines several models to reduce generalization error (see Appendix)
  1. train different models separately
  2. then have all models vote on output for test examples

- how to "train different models"?
  - construct k different datasets by random sampling
  - reuse same model/training algorithm/objective function

  $\Rightarrow$ equivalent to training different models

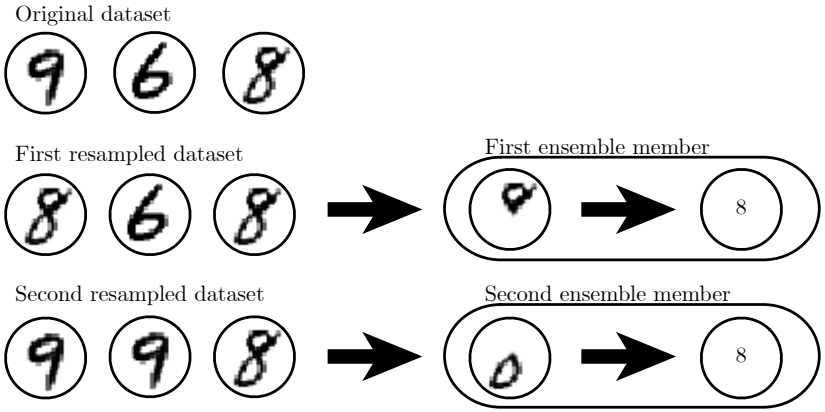


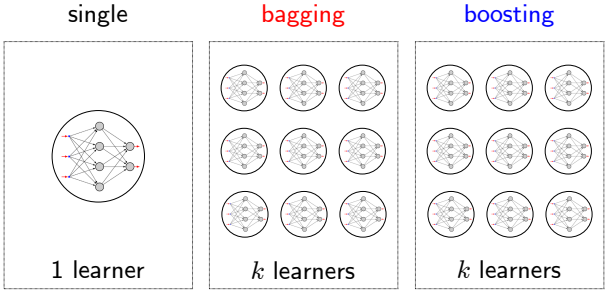
image source: I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016

Boosting

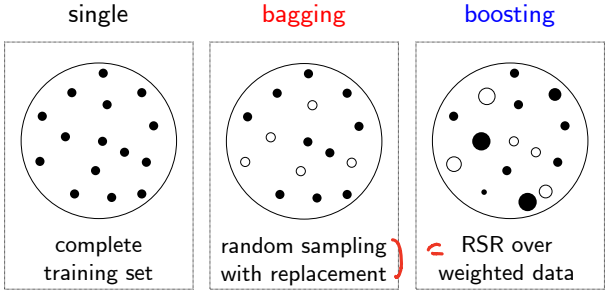
- constructs an ensemble with higher capacity than individual models
  - meta-algorithm for primarily reducing bias, and also variance
  - most famous: AdaBoost
- train multiple weak learners sequentially to get a strong leaner
  - future learners focus more on the examples previous learners misclassified
  - how? reweight training examples wrt previous learning results
- boosting examples in neural nets:
  - incrementally add neural nets to the ensemble
  - incrementally add hidden units to the neural net

Comparison

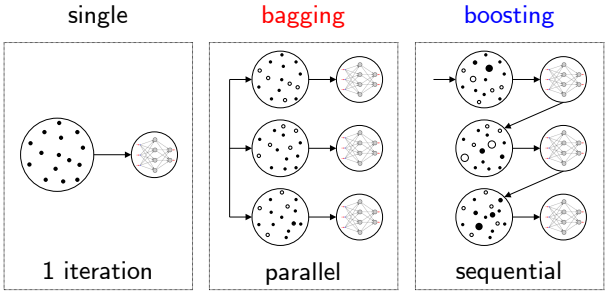
- # of leaner(s)



- data



- training



- aggregation[4]

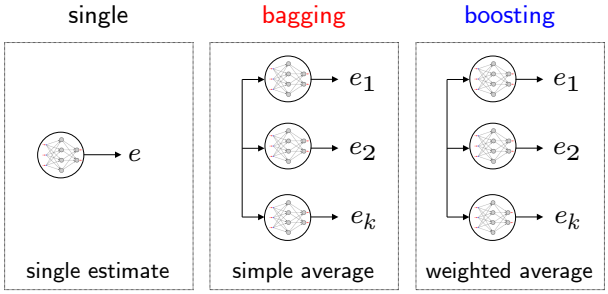


image source: https://quantdare.com/what-is-the-difference-between-bagging-and-boosting

[4] simple average: $e = \frac{1}{k}\sum_{i=1}^{k} e_i$, weighted average: $e = \frac{1}{k}\sum_{i=1}^{k} w_i e_i$

Neural networks

- reach a wide enough variety of solution points
  - that they can often benefit from model averaging
  - even if all the models are trained on the same dataset
- differences in
  - random initialization
  - random selection of minibatches
  - hyperparameters
  - non-deterministic implementations

$\Rightarrow$ often enough to cause

  - different members of the ensemble to make partially independent errors

Final remarks

- model ensembles: extremely powerful and reliable to reduce generalization error
  1. train multiple independent models
  2. at test time: average their results

  $\Rightarrow$ typically gives about 2% extra performance (price: computation/memory)

- ML contests:
  - won by methods using model averaging over dozens of models

  e.g. Netflix Grand Prize, Kaggle, and many more

c.f. scientific papers: ensembles discouraged when benchmarking algorithms

- benchmark comparisons: usually made using a single model

Outline

Dropout

- computationally inexpensive but powerful regularization
  - makes bagging practical for large neural nets
- recall: bagging
  - trains multiple models and
  - evaluates multiple models on each test example

  $\Rightarrow$ impractical when each model is a large neural net
- common: use ensembles of five to ten neural nets

  e.g. Szegedy *et al.* (2014a) used six to win ILSVRC
  - but more than this rapidly becomes unwieldy
- dropout provides an inexpensive approximation to
  - training/evaluating a bagged ensemble of exponentially many neural nets

Idea

- in each forward pass
  - randomly set some neurons to zero
- probability of inclusion (or dropping out)
  - a hyperparameter (commonly: 0.5 for hidden units)

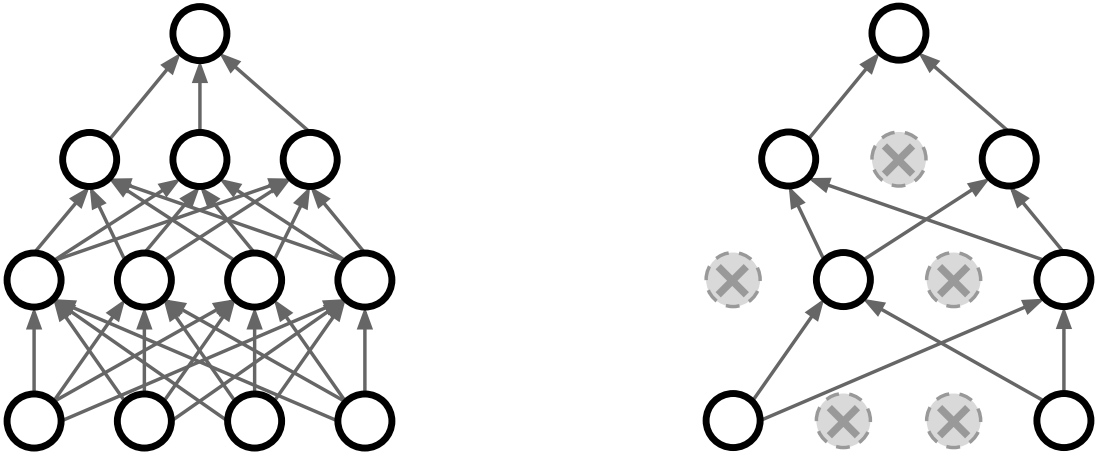
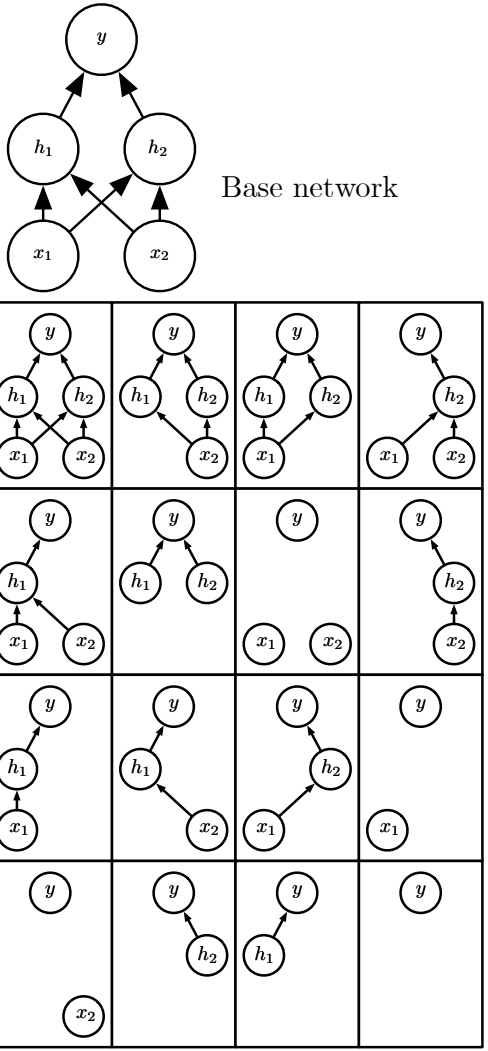


image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017), http://cs231n.stanford.edu/2017/index.html

Example



Base network

Ensemble of subnetworks

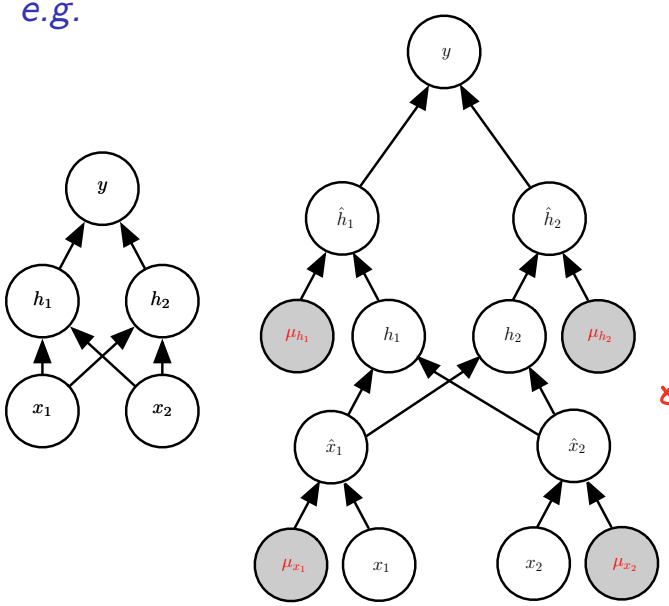
- given an underlying base network
  - *e.g.* two input units and two hidden units
    - $\Rightarrow$ sixteen subnetworks
- dropout trains the ensemble of all subnetworks
- subnetworks: formed by
  - removing non-output units from base net
- how to remove a unit?
  - multiply its output value by 0

image source: I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016

Dropout: training time

- each time we load an example into a minibatch
  - randomly sample a binary mask[5] to apply to all input/hidden units
- then run as usual: forward prop $\rightarrow$ backprop $\rightarrow$ parameter updates

e.g.



- let $\boldsymbol{\mu}$: mask vector
  - specifies which units to include
- top right subnet on page 39

$$\boldsymbol{\mu} = (\mu_{x_1}, \mu_{x_2}, \mu_{h_1}, \mu_{h_2}) = (1, 1, 0, 1)$$

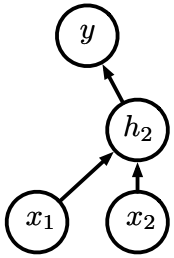


image source: I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016

[5] typically, prob{mask value = 1}: 0.5 (hidden unit), 0.8 (input unit)

Dropout: inference time

- ensemble prediction (bagging):

$$p_{\text{ensemble}}(y \mid \boldsymbol{x}) = \frac{1}{k} \sum_{i=1}^{k} \underbrace{p^{(i)}(y \mid \boldsymbol{x})}_{\text{from model } i} \tag{2}$$

- ensemble prediction (dropout):

$$p_{\text{ensemble}}(y \mid \boldsymbol{x}) = \sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) \underbrace{p(y \mid \boldsymbol{x}, \boldsymbol{\mu})}_{\substack{\text{submodel defined by} \\ \text{mask vector } \boldsymbol{\mu}}} \tag{3}$$

  - $p(\boldsymbol{\mu})$: probability distribution used to sample $\boldsymbol{\mu}$ at training time
  - eq (3): intractable ($\leftarrow$ exponential number of terms in $\sum$) (cμ)
    - solution: just average over many masks (*e.g.* 10–20) or
    - approximate $p_{\text{ensemble}}(y \mid \boldsymbol{x})$ by evaluating $p(y \mid \boldsymbol{x})$ in one model (*e.g.* weight scaling inference rule)

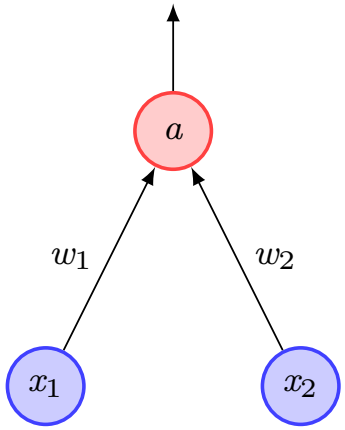
Weight scaling inference rule (Hinton *et al.*, 2012)

- for each unit:
  - replace outgoing weight $\boldsymbol{W} \longrightarrow \alpha W$ (if α : dropping rate $\rightarrow (1-\alpha)W$)
  - α: probability of including (not dropping out) this unit
- motivation
  - at test time: all neurons are always active

  $\Rightarrow$ we must scale activations so that for each neuron:

  output at test time $=$ expected output at training time

- example: a simple model ($\alpha = 0.5$)



  - test time:

$$a = w_1 x_1 + w_2 x_2$$

  - training time:

$$\begin{aligned}\mathbb{E}[a] &= \frac{1}{4}(w_1 x_1 + w_2 x_2) + \frac{1}{4}(w_1 x_1 + 0) \\ &\quad + \frac{1}{4}(0 + w_2 x_2) + \frac{1}{4}(0 + 0) \\ &= \frac{1}{2}(w_1 x_1 + w_2 x_2)\end{aligned}$$

  - thus $\underbrace{\frac{1}{2}a}_{\substack{\text{adjusted output} \\ \text{at test time}}} = \underbrace{\mathbb{E}[a]}_{\substack{\text{expected output} \\ \text{at traing time}}}$

- bottom line: at test time, multiply by inclusion probability α

- weight scaling rule with usual inclusion probability (0.5)
  1. training time: dropout with prob 0.5
  2. test time: divide activations by 2

- inverted dropout: more common way to achieve the same result
  1. training time: dropout with prob 0.5 and then multiply activations by 2
  2. test time: no change ($\rightarrow$ computationally more efficient)

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

drop in fwd pass

scale at test time

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time: no change

image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017), http://cs231n.stanford.edu/2017/index.html

Dropout advantages

1. more effective than standard computationally inexpensive regularizers

   e.g. weight decay, filter norm constraints, sparse activity regularization

   - may also be combined with other forms of regularization

2. very computationally cheap

   - both training and inference are efficient

3. works well with nearly any model

   - that uses a distributed representation and can be trained with SGD

   e.g. feedforward nets, RBM, RNN

Cost and limitations

cost for using dropout:

- much larger model and many more iterations of training algorithm

  dropout = regularizer

  $\Rightarrow$ reduces the effective capacity of a model

  $\Rightarrow$ to offset this effect, we must increase model size

when is dropout less effective?

1. very large datasets:
   - regularization confers little reduction in generalization error
   - Computational cost may outweigh benefit of regularization
2. extremely few labeled training examples are available

Related work

- extensions/improvements
  - fast dropout
    - analytical approximation to the sum over all submodels
  - dropout boosting
    - dropout : bagging = dropout boosting : boosting
    - not for regularization but for jointly maximizing likelihood
- implicit ensemble methods (consider exponentially large ensembles)
  - stochastic pooling: builds ensembles of conv nets
  - DropConnect: sets a random subset of weights to zero

  c.f. dropout: randomly sets hidden units to zero

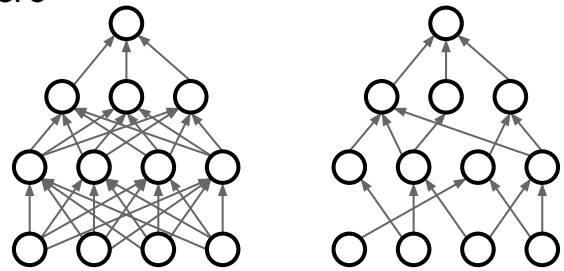


image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017), http://cs231n.stanford.edu/2017/index.html

Why does dropout work?

- dropout training $\Rightarrow$ a model cannot rely on any one feature
  - $\Rightarrow$ should spread out weights $\Rightarrow$ shrink weights
- dropout trains an ensemble of models sharing hidden units
  - $\Rightarrow$ each hidden unit: regularized to encode a feature good in many contexts
- masking noise: applied to hidden units[6]
  - highly intelligent/adaptive destruction of input information

e.g. if a hidden unit h_i detects a face by finding eyebrows

- dropping h_i = erasing the info that there are eyebrows
- the model must learn another h_i that either
  - redundantly encodes the presence of eyebrows, or
  - detects the face by another feature (*e.g.* mouth)

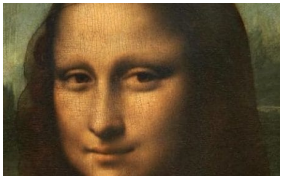


image modified from: https://images.app.goo.gl/UaAiooK7TGvhuhELA

[6] a large portion of the power of dropout arises from this fact

Batch normalization

- reparametrizes a model to introduce
  - both additive and multiplicative noise on hidden units

  c.f. dropout: only multiplicative noise on hidden units

- primary purpose: better optimization
  - but noise can have a regularizing effect

  $\Rightarrow$ sometimes makes dropout unnecessary

image source: https://images.app.goo.gl/RZm6ZAzUnzAuXdhx5

Outline

Theme: immunizing a model by noise injection

- various injection points exist:

noise added to	technique	reference
input	(1) data augmentation (2) imposing penalty on weight norm	sec 7.4 Bishop (1995a, b)
hidden units	dropout	sec 7.12
weights	(1) Bayesian inference over weights (2) stabilizing learned function	sec 7.5 sec 7.5
output targets	label smoothing	sec 7.5

- label smoothing: assume y is correct with prob $1-\epsilon$ (ϵ : small constant)
  - regularizes a model based on a softmax with k output values
    - (hard) label $0 \rightarrow$ soft label $\frac{\epsilon}{k-1}$
    - (hard) label $1 \rightarrow$ soft label $1-\epsilon$

image source: https://images.app.goo.gl/jvUP9YEmh6o5Pj1k8

- another common pattern
  - training: add random noise
  - testing: marginalize over the noise

- examples
  - dropout
  - DropConnect
  - data augmentation
  - batch norm
  - stochastic depth

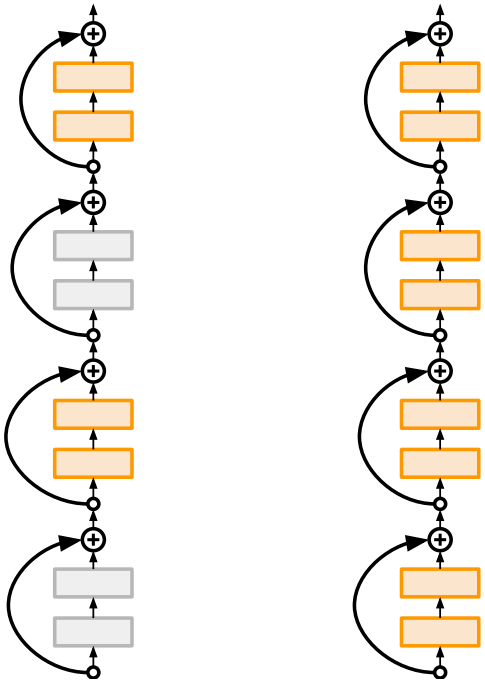
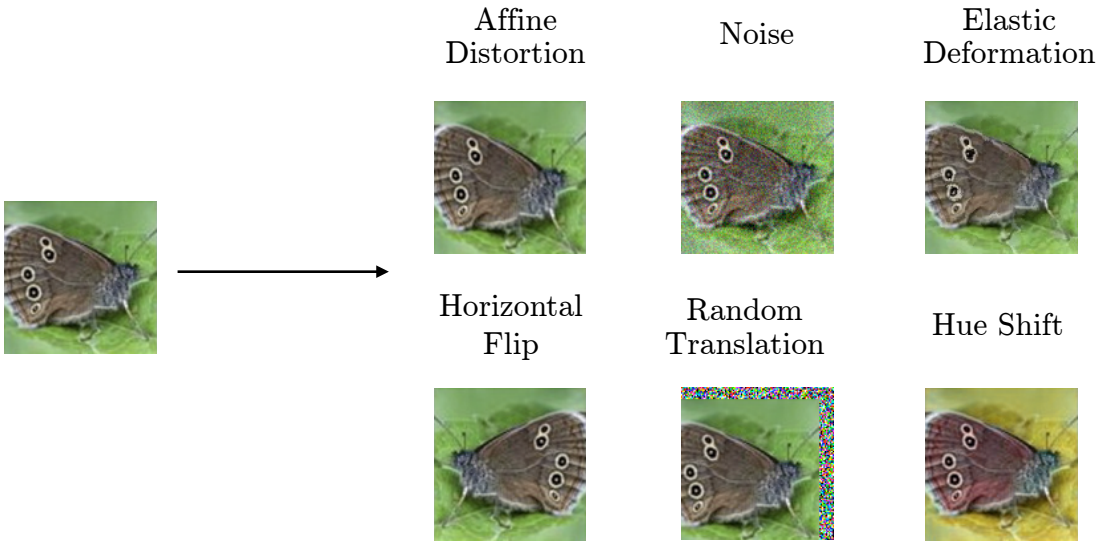


image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017), http://cs231n.stanford.edu/2017/index.html

Data augmentation

- create fake data and add it to training set



- difficulty of creating fake data: depends on specific ML tasks
  - successful cases reported: object recognition, speech recognition

image source: https://www.deeplearningbook.org/slides/07_regularization.pdf

random crops and scales (ResNet):

- **training**: sample random crops/scales
  1. pick random $L \in [256, 480]$
  2. resize training image (short side $= L$)
  3. sample random 224×224 patch

- **testing**: average results from a fixed set of crops
  1. resize image at five scales: {224, 256, 384, 480, 640}
  2. for each size, use ten 224×224 crops: (center + 4 corners) x flips

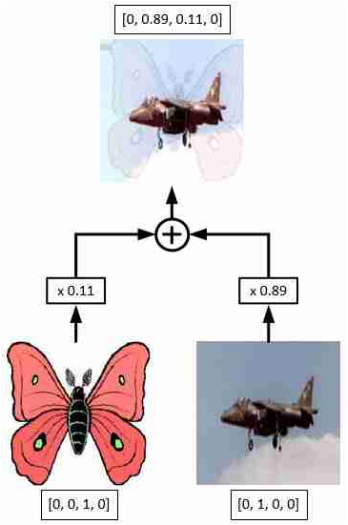
image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017), http://cs231n.stanford.edu/2017/index.html; Ng, *Deep Learning* (Coursera), https://www.coursera.org/specializations/deep-learning

Mixup[7]

- trains a neural net on convex combinations of pairs of examples and their labels

$$\tilde{\boldsymbol{x}} = \lambda \boldsymbol{x}_i + (1-\lambda)\boldsymbol{x}_j$$
$$\tilde{y} = \lambda y_i + (1-\lambda) y_j$$

  - $\boldsymbol{x}_i, \boldsymbol{x}_j$: raw input vectors
  - y_i, y_j : one-hot label encodings



- prior knowledge incorporated:
  - linear interpolations of feature vectors

  $\Rightarrow$ linear interpolations of associated targets

- mixup regularizes the neural net to
  - favor *simple* linear behavior in-between training examples

  $\Rightarrow$ can improve the generalization of (even sota) neural nets

image source: https://www.dlology.com/blog/how-to-do-mixup-training-from-image-files-in-keras/

[7] Hongyi Zhang et al. "mixup: Beyond empirical risk minimization". In: *arXiv preprint arXiv:1710.09412* (2017)

comparison:

- Mixup (Zhang *et al.*, 2017)
  - mix two samples by linear interpolation of images and labels
- Cutout (DeVries and Taylor, 2017)
  - zero out a random patch from an image (regional dropout)
- CutMix (Yun *et al.*, 2019)
  - cut and paste random patches between training images

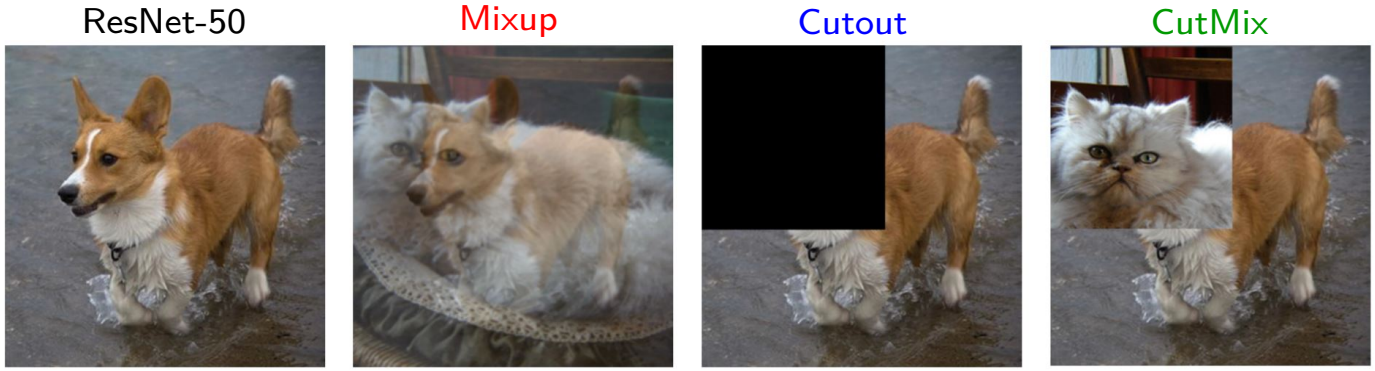


image source: https://arxiv.org/abs/1905.04899

Multi-task learning

- method #1: pool training examples out of several tasks
  - additional examples $\approx$ soft constraints imposed on parameters

  $\Rightarrow$ put more pressure on parameters towards values that generalize well

  $\Rightarrow$ better generalization

- method #2: share part of a model across tasks
  - shared part: more constrained towards good values

  $\Rightarrow$ better generalization

  e.g. $\boldsymbol{h}^{(\text{shared})}$ captures common factors

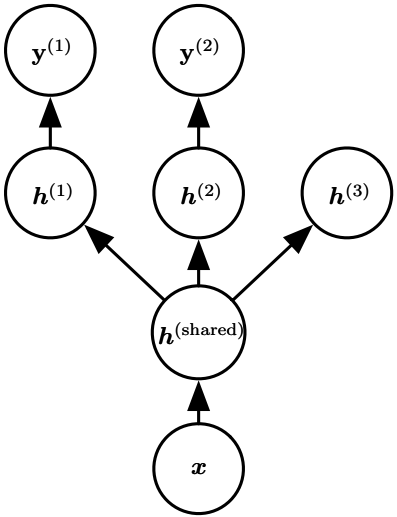


image source: I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016

Parameter tying and parameter sharing

- parameter tying
  - prior: similar tasks would have similar parameter values
  - leverage this information through regularization

  e.g. parameter norm penalty (for models A and B):

$$\Omega\left(\boldsymbol{w}^{(A)}, \boldsymbol{w}^{(B)}\right) = ||\boldsymbol{w}^{(A)} - \boldsymbol{w}^{(B)}||_2^2$$

- parameter sharing (more popular)
  - make sets of parameters equal

  $\Rightarrow$ vast representational (*i.e.* fewer weights)/computational efficiency

  e.g. convolutional neural nets (ch 9)
    - dramatically lower # of parameters
    - significantly increase net sizes w/o a corresponding increase in data

Sparse representations

- place penalty on activations of units encouraging sparse representations
  - *e.g.* L^1 penalty on hidden representation: $\Omega(\boldsymbol{h}) = ||\boldsymbol{h}||_1 = \sum_i |h_i|$
    - $\Rightarrow$ implicitly imposes a complicated penalty on parameters
  - c.f. lasso $\Omega(\boldsymbol{w}) = ||\boldsymbol{w}||_1$: places penalty directly on parameters

- example: simplified view in linear regression

$$\underset{\boldsymbol{y}\in\mathbb{R}^m}{\begin{bmatrix}18\\5\\15\\-9\\-3\end{bmatrix}} = \underset{\boldsymbol{A}\in\mathbb{R}^{m\times n}}{\begin{bmatrix}4&0&0&-2&0&0\\0&0&-1&0&3&0\\0&5&0&0&0&0\\1&0&0&-1&0&-4\\1&0&0&0&-5&0\end{bmatrix}} \underset{\boldsymbol{x}\in\mathbb{R}^n}{\begin{bmatrix}2\\3\\-2\\-5\\1\\4\end{bmatrix}}$$

(sparse parameterization)
→ Lasso

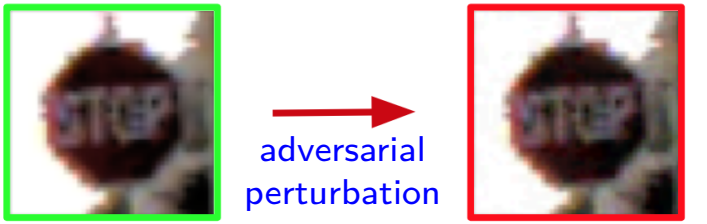
$$\underset{\boldsymbol{y}\in\mathbb{R}^m}{\begin{bmatrix}-14\\1\\19\\2\\23\end{bmatrix}} = \underset{\boldsymbol{B}\in\mathbb{R}^{m\times n}}{\begin{bmatrix}3&-1&2&-5&4&1\\4&2&-3&-1&1&3\\-1&5&4&2&-3&-2\\3&1&2&-3&0&-3\\-5&4&-2&2&-5&-1\end{bmatrix}} \underset{\boldsymbol{h}\in\mathbb{R}^n}{\begin{bmatrix}0\\2\\0\\0\\-3\\0\end{bmatrix}}$$

(sparse representation)

Outline

Adversarial examples

- adversarially perturbed examples
  - even neural nets with human-level performance make ∼100% error



stop sign
(correct)

max speed 100
(misclassified)

- how to create?
  - intentionally constructed using optimization
    - search for input $\boldsymbol{x}'$ near data point $\boldsymbol{x}$ s.t. their labels differ

i.e. $\boldsymbol{x}' \approx \boldsymbol{x}$ but $y(\boldsymbol{x}') \neq y(\boldsymbol{x})$

image source:
https://mlatgt.blog/2018/08/09/shield-defending-deep-neural-networks-from-adversarial-attacks/

- concept:
  - adversarial examples change decision boundary

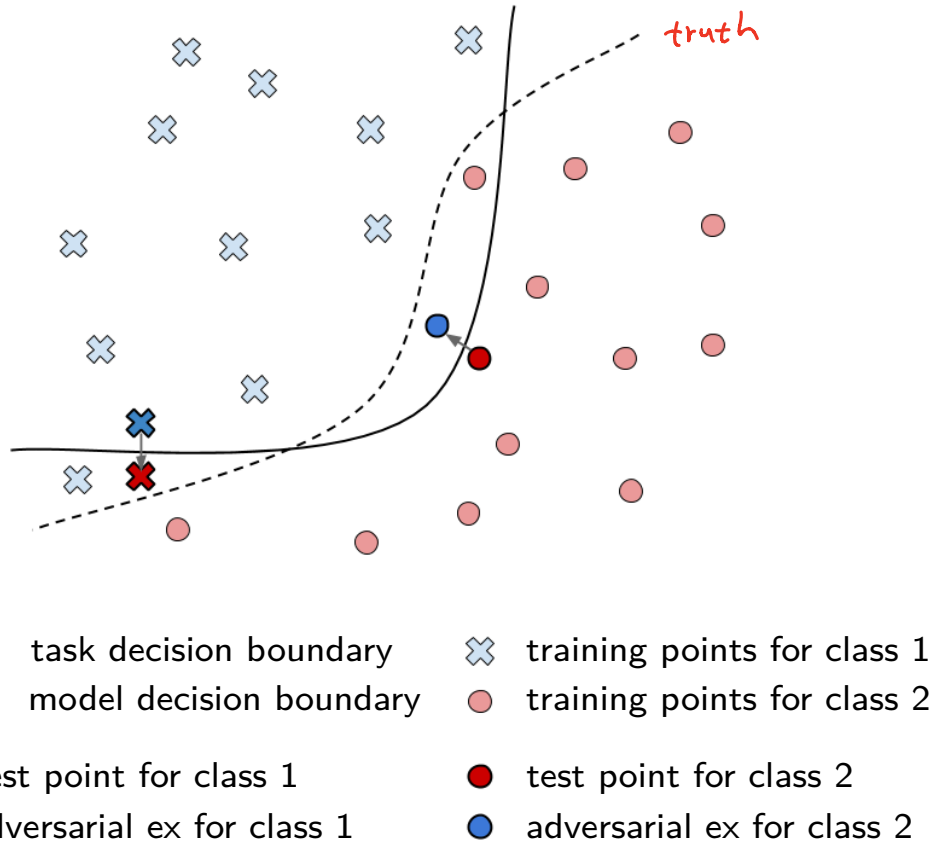
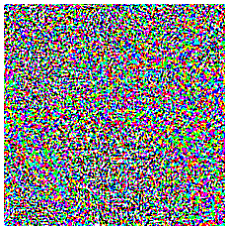


image source: blog by Goodfellow and Papernot (http://www.cleverhans.io)

- adversarial example generation applied to GoogLeNet on ImageNet:


$+ .007 \times$

$=$

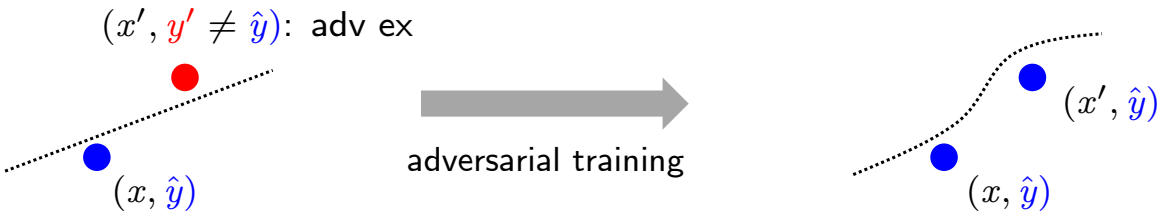

$\boldsymbol{x}$	$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$	$\boldsymbol{x} + \epsilon\,\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
$y =$ "panda"	"nematode"	"gibbon"
w/ 57.7%	w/ 8.2%	w/ 99.3 %
confidence	confidence	confidence

- change GoogLeNet's classification by adding imperceptibly small vector
  - its elements = sign of the elements of the gradient of cost function wrt input
  - called "fast gradient sign method"

image source: Goodfellow, I.J., Shlens, J. and Szegedy, C., 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*.

Adversarial training

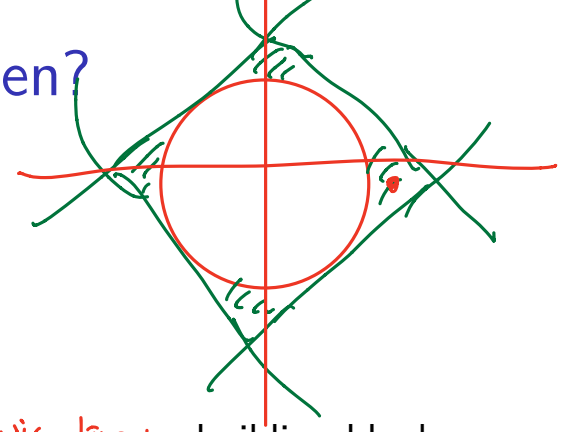
- training with adversarial examples x'
  1. find inputs x' near the original inputs x
  2. train model to produce $y(x') = y(x)$



  - based on the manifold assumption

- two desirable effects
  1. can reduce error on original test set
     $\Rightarrow$ regularization effect
  2. can give correct labels to data points with incorrect labels
     $\Rightarrow$ semi-supervised learning effect

Why do adversarial examples happen?



- one of the primary causes:
  - excessive linearity
- modern neural nets: built out of primarily piecewise linear building blocks

  $\Rightarrow$ overall function they implement: sometimes highly linear

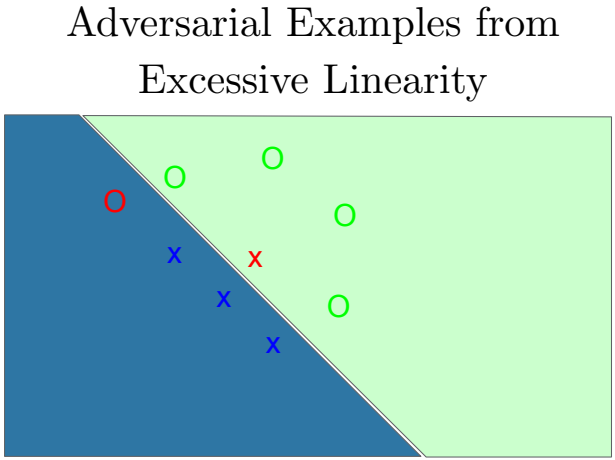
- unfortunately, a linear function
  - can change very rapidly if it has numerous inputs (*i.e.* high dim)

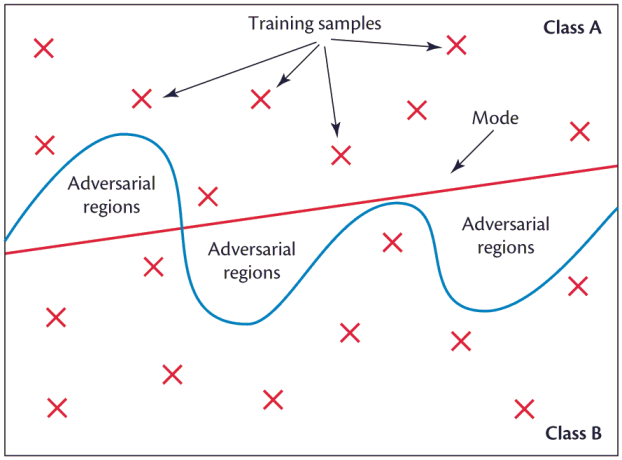
  i.e. if we change each input by ϵ

  $\triangleright$ a linear function with weights $\boldsymbol{w}$ can change by as much as $\underbrace{\epsilon||\boldsymbol{w}||_1}_{\uparrow}$

  very large if $\boldsymbol{w}$ is high-dimensional

- Goodfellow *et al.* (2014b):
  - adversarial examples are from underfitting (rather than from overfitting)





- adversarial training
  - discourages this highly sensitive locally linear behavior by
  - encouraging net to be locally constant in neighborhood of training data

  $=$ explicitly introducing a local constancy prior into supervised neural nets

image sources: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017), `http://cs231n.stanford.edu/2017/index.html`; McDaniel, P., Papernot, N. and Celik, Z.B., 2016. Machine learning in adversarial settings. *IEEE Security & Privacy*, 14(3), pp.68-72.

Outline

Summary

- regularization: constraining a model not to fit noise (a must in practice)
  - effective when stochastic/deterministic noise is present
- ways to constrain a model (toward simpler/smoother direction)
  - use augmented error: better proxy for $\mathrm{E}_{\mathrm{gen}}$ than $\mathrm{E}_{\mathrm{train}}$

$$\mathrm{E}_{\mathrm{aug}}(\boldsymbol{w}) = \mathrm{E}_{\mathrm{train}}(\boldsymbol{w}) + \underbrace{\overbrace{\frac{\lambda}{N}}^{\text{regularization parameter}} \overbrace{\Omega(\boldsymbol{w})}^{\text{regularizer}}}_{\text{penalty term}}$$

  - noise added to input/hidden units (dropout)/weights/output target
  - multi-task learning, parameter sharing, early stopping, ensembling
- choosing $\Omega(\boldsymbol{w})$: heuristic (popular: l_1 lasso, l_2 ridge ["weight decay"])
- selecting regularization parameter λ: by validation ($\lambda = 0$ for wrong Ω)

Outline

Regularization and VC theory

regularization by constrained-minimizing $\mathrm{E}_{\text{train}}$

$$\min_{\mathbf{w}} \mathrm{E}_{\text{train}}(\mathbf{w}) \text{ s.t. } \mathbf{w}^\top \mathbf{w} \leq C$$

$C \downarrow$

$\Longleftrightarrow$

gen $\uparrow$

VC guarantee of constrained-minimizing $\mathrm{E}_{\text{train}}$

$$\mathrm{E}_{\text{test}}(\mathbf{w}) \leq \mathrm{E}_{\text{train}}(\mathbf{w}) + \Omega(\mathcal{H}(C))$$

$\Updownarrow$ C equivalent to some λ

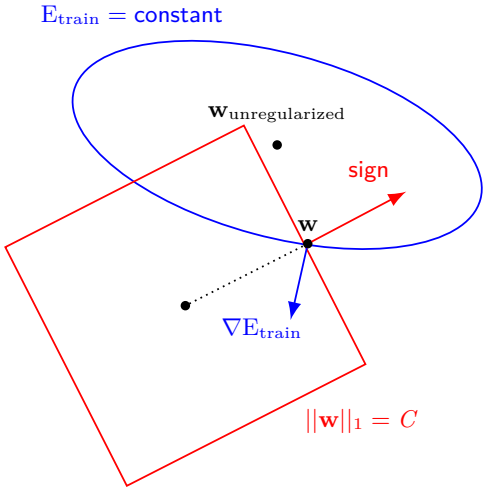
regularization by minimizing E_{aug}

$$\min_{\mathbf{w}} \left\{ \mathrm{E}_{\text{aug}}(\mathbf{w}) = \mathrm{E}_{\text{train}}(\mathbf{w}) + \frac{\lambda}{N} \mathbf{w}^\top \mathbf{w} \right\}$$

- minimizing E_{aug}
  - indirectly getting VC guarantee without confining to $\mathcal{H}(C)$

Comparison: l_1 and l_2 regularization

l_1 (lasso)



l_2 (ridge regression)

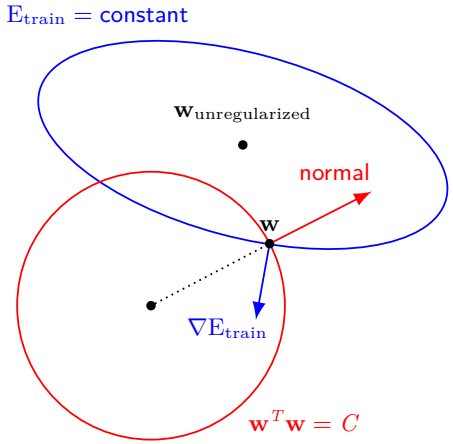


image source: Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from Data*. AMLBook, 2012

What if we pick a wrong regularizer?

- *e.g.* weight *growth*
  - $\Omega(\mathbf{w}) = \sum_{q=0}^{Q} \frac{1}{w_q^2}$
- don't worry:
  - we still have λ
  - *validation* will set $\lambda = 0$

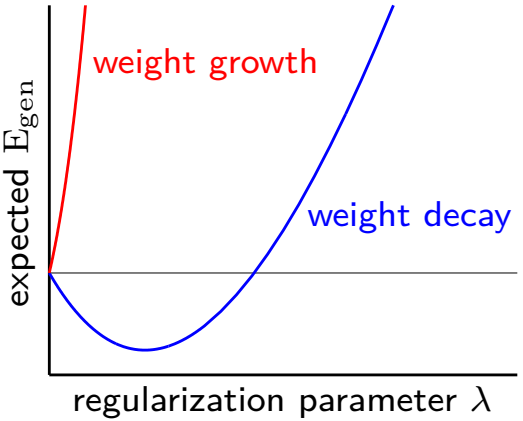


image source: Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from Data*. AMLBook, 2012

Ensemble example: a set of k regression models

- suppose: each model makes an error ϵ_i on each example
  - errors: drawn from a zero-mean multivariate normal distribution
    - variances $E[\epsilon_i^2] = v$, covariances $E[\epsilon_i \epsilon_j] = c$

- error (made by average prediction of all the ensemble models):

$$\frac{1}{k} \sum_i \epsilon_i$$

- expected squared error of the ensemble predictor:

$$\begin{aligned} \mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] &= \frac{1}{k^2}\mathbb{E}\left[\sum_i \left(\epsilon_i^2 + \sum_{j \neq i} \epsilon_i \epsilon_j\right)\right] \\ &= \frac{1}{k} v + \frac{k-1}{k} c \end{aligned}$$

- if errors are perfectly correlated and $c = v$
  - mean squared error reduces to v

  $\Rightarrow$ model averaging does not help at all

- if errors are perfectly uncorrelated and $c = 0$
  - expected squared error of the ensemble: only $\frac{1}{k}v$

  $\Rightarrow$ decreases linearly with the ensemble size

- in other words: on average, the ensemble will perform
  - at least as well as any of its members and
  - if the members make independent errors

  $\Rightarrow$ the ensemble will perform significantly better than its members