

# CS5339 Machine Learning

## Optimization II

Lee Wee Sun  
School of Computing  
National University of Singapore  
leews@comp.nus.edu.sg

Semester 2, 2019/20

# Outline

- 1 Deep Learning
- 2 Boosting and Decision Trees
- 3 EM/K-means
- 4 Appendix

# Gradient Descent

- Learning for deep learning is currently mostly done using (mini-batch) stochastic gradient descent.
- Approximate second-order methods possible but harder to get to work reliably.
  - Higher precision may not be necessary in learning.
  - Saddle points may be problematic.

# Back-Propagation: Computing Gradient for Deep Learning

- For simplicity, we consider the regular network structure, where the output of layer  $t$  is connected only to the next layer  $t + 1$  and not to deeper layers.
- Consider squared loss, although easy to change to other loss functions.
- Often we add a regularization term  $\frac{\lambda}{2} \mathbf{w}^T \mathbf{w}$ . Gradient of this with respect to  $w_i$  is just  $\lambda w_i$ . In neural networks literature, this regularization is often called weight decay.

## Backpropagation

input:

example  $(\mathbf{x}, \mathbf{y})$ , weight vector  $\mathbf{w}$ , layered graph  $(V, E)$ ,  
 activation function  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$

initialize:

denote layers of the graph  $V_0, \dots, V_T$  where  $V_t = \{v_{t,1}, \dots, v_{t,k_t}\}$   
 define  $W_{t,i,j}$  as the weight of  $(v_{t,j}, v_{t+1,i})$   
 (where we set  $W_{t,i,j} = 0$  if  $(v_{t,j}, v_{t+1,i}) \notin E$ )

forward:

set  $\mathbf{o}_0 = \mathbf{x}$   
 for  $t = 1, \dots, T$   
   for  $i = 1, \dots, k_t$   
     set  $\underline{a}_{t,i} = \sum_{j=1}^{k_{t-1}} W_{t-1,i,j} \underline{o}_{t-1,j}$   
     set  $\underline{o}_{t,i} = \sigma(\underline{a}_{t,i})$

compute activation  
& outputs  
in forward pass

backward:

set  $\delta_T = \mathbf{o}_T - \mathbf{y}$   
 for  $t = T-1, T-2, \dots, 1$   
   for  $i = 1, \dots, k_t$   
      $\delta_{t,i} = \sum_{j=1}^{k_{t+1}} W_{t+1,i,j} \delta_{t+1,j} \sigma'(\underline{a}_{t+1,j})$

compute  $\delta$   
for all units  
backwards

output:

foreach edge  $(v_{t-1,j}, v_{t,i}) \in E$   
 set the partial derivative to  $\delta_{t,i} \sigma'(\underline{a}_{t,i}) \underline{o}_{t-1,j}$

go through  
every edge

# Derivation of Backpropagation

We will first describe the chain rule of calculus.

- Let  $x$  be a real number and  $f$  and  $g$  are both functions mapping from a real to a real number. Suppose  $y = g(x)$  and  $z = f(g(x)) = f(y)$ . Then

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}.$$

- Suppose that  $\mathbf{x} \in \mathbb{R}^m$ ,  $\mathbf{y} \in \mathbb{R}^n$ ,  $g$  maps from  $\mathbb{R}^m$  to  $\mathbb{R}^n$  and  $f$  maps from  $\mathbb{R}^n$  to  $\mathbb{R}$ . If  $\mathbf{y} = g(\mathbf{x})$  and  $z = f(\mathbf{y})$ , then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}.$$

$\mathbf{x} \in \mathbb{R}^m$   
 $\mathbf{y} \in \mathbb{R}^n$   $\mathbf{y} = (y_1(x), \dots, y_n(x))$   
 $z \in \mathbb{R}$   $z = f(y_1(x), \dots, y_n(x))$

We assume that all weights are present. If not present, just fix to zero.

- We begin with the output layer weight and assume  $k_T$  outputs with the squared loss  $\ell(\mathbf{y}, \mathbf{o}_T) = \frac{1}{2} \sum_{i=1}^{k_T} (y_i - o_{T,i})^2$  where  $\mathbf{o}_t$  is the output of nodes at layer  $t$  and  $T$  is the final layer. Let

$$\underline{\delta_{T,i}} = \frac{\partial}{\partial o_{T,i}} \frac{1}{2} (y_i - o_{T,i})^2 = \underline{(o_{T,i} - y_i)},$$

$$\underline{\sigma'(a_{T,i})} = \frac{\partial o_{T,i}}{\partial a_{T,i}} = \underline{o_{T,i}(1 - o_{T,i})} \text{ when } \sigma(a_{T,i}) = \frac{1}{1 + \exp(-a_{T,i})}.$$

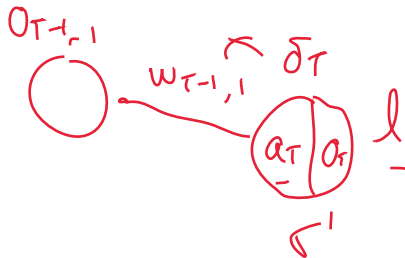


- We also have

$$\frac{\partial \underline{a_{T,i}}}{\partial \underline{w_{T-1,i,j}}} = \underline{o_{T-1,j}}.$$

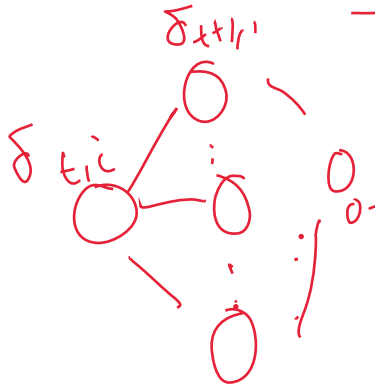
Then

$$\begin{aligned} \frac{\partial \underline{\ell}}{\partial \underline{w_{T-1,i,j}}} &= \frac{\partial \underline{\ell}}{\partial \underline{o_{T,i}}} \frac{\partial \underline{o_{T,i}}}{\partial \underline{a_{T,i}}} \frac{\partial \underline{a_{T,i}}}{\partial \underline{w_{T-1,i,j}}} \\ &= \underline{\delta_{T,i}} \underline{\sigma'(a_{T,i})} \underline{o_{T-1,j}}. \end{aligned}$$





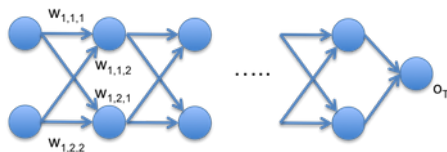
- For computing  $\frac{\partial \ell}{\partial w_{t-1,i,j}}$ , we again compute



$$\begin{aligned}
 \underline{\delta_{t,i}} &= \frac{\partial \underline{\ell}}{\partial \underline{o_{t,i}}} \\
 &= \sum_{j=1}^{k_{t+1}} \frac{\partial \underline{\ell}}{\partial \underline{o_{t+1,j}}} \frac{\partial \underline{o_{t+1,j}}}{\partial \underline{o_{t,i}}} \\
 &= \sum_{j=1}^{k_{t+1}} \underline{\delta_{t+1,j}} \frac{\partial \underline{o_{t+1,j}}}{\partial \underline{a_{t+1,j}}} \frac{\partial \underline{a_{t+1,j}}}{\partial \underline{o_{t,i}}} \\
 &= \sum_{j=1}^{k_{t+1}} \underline{\delta_{t+1,j}} \sigma'(\underline{a_{t+1,j}}) \underline{w_{t,j,i}}.
 \end{aligned}$$

- Having computed all  $\delta_{t,i}$ , we can then compute

$$\begin{aligned}\frac{\partial \ell}{\partial w_{t-1,i,j}} &= \frac{\partial \ell}{\partial o_{t,i}} \frac{\partial o_{t,i}}{\partial a_{t,i}} \frac{\partial a_{t,i}}{\partial w_{t-1,i,j}} \\ &= \delta_{t,i} \sigma'(a_{t,i}) o_{t-1,j}.\end{aligned}$$



### Exercise O2-1 (Archipelago):

Consider the following simple recursive algorithm for computing the  $\delta_{t,i}$  value required for backpropagation.

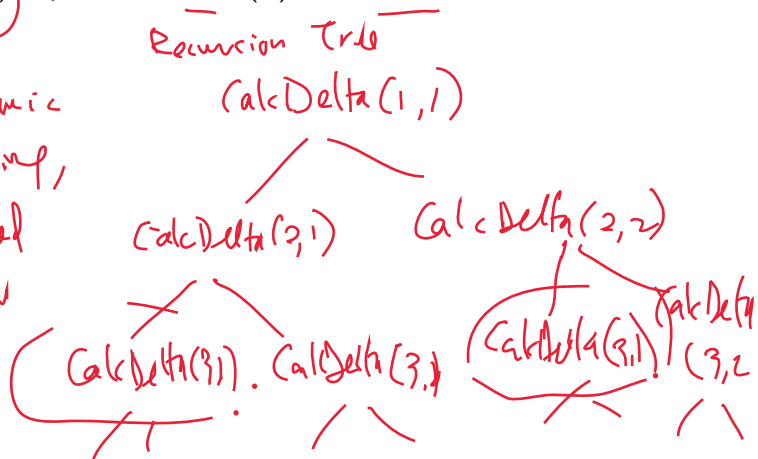
CALCDELTA( $t, i$ )

```

1  if  $t == T$ 
2      return  $(o_{T,i} - y_i)$ 
3  curr = 0
4  for  $j = 1$  to  $k_{t+1}$ 
5      curr = curr + CALCDELTA( $t+1, j$ )  $\times \sigma'(a_{t+1, j}) \times w_{t, j, i}$ 
6  return curr
```

What is the runtime when CALCDELTA is applied to the network shown? (A) Exponential in  $T$ . (B) Linear in  $T$ .

Backprop  
does dynamic  
programming,  
reuse stored  
computation



# Pitfalls

- III-Conditioning

- Recall Taylor approximation

$$f(\mathbf{w}^{(0)} - \eta \mathbf{g}) \approx f(\mathbf{w}^{(0)}) - \eta \mathbf{g}^T \mathbf{g} + \frac{1}{2} \eta^2 \mathbf{g}^T \mathbf{H} \mathbf{g}.$$

- When  $\frac{1}{2} \eta^2 \mathbf{g}^T \mathbf{H} \mathbf{g}$  exceed  $\eta \mathbf{g}^T \mathbf{g}$ , objective function increases.
- When problem is ill-conditioned, even very small step size increases objective function.
- Learning becomes very slow.
- Newton's method hard to use in neural networks.

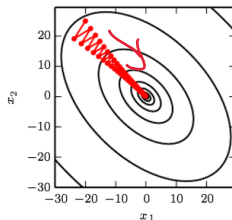


Figure from [2].

- Local minimum

- Neural networks can have a large number of local minimas.
- But only local minimas with high objective functions are a problem.
  - For years, practitioners believe these are common.
  - But now, possibly not, although still not clear.



- Check the norm of the gradient over time to diagnose – if does not shrink, probably not local minima.

- Plateaus, saddle points, flat regions.
  - Saddle points may be quite common.
  - Gradient can become very small near saddle point.
  - Unclear whether it is a bad problem in practice.
  - Wide flat regions of constant value is also a problem.

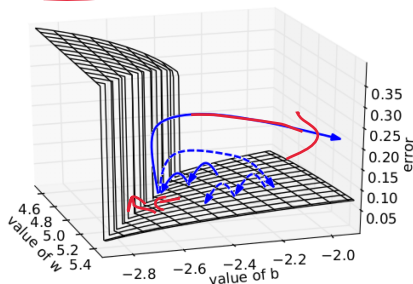


Figure from [5].

- Cliffs and exploding gradients
  - Very large gradient. Gradient update move parameter to different region.
  - If gradient too large, rescale or truncate.

- Vanishing/exploding gradient in recurrent neural networks

- Gradient of loss at time  $i$  with respect to hidden layer  $j$

$$\begin{aligned}\frac{\partial \ell^{(i)}}{\partial \mathbf{h}^{(j)}} &= \frac{\partial \ell^{(i)}}{\partial \mathbf{h}^{(i)}} \prod_{j < t \leq i} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} \\ &= \frac{\partial \ell^{(i)}}{\partial \mathbf{h}^{(i)}} \prod_{j < t \leq i} \mathbf{W}_h \text{diag} \left( \sigma' \left( \mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b} \right) \right)\end{aligned}$$

Use  $\|AB\| \leq \|A\| \|B\|$   
 $\|A\|$  is largest singular val

- With matrix 2-norm (spectral norm)

$$\left\| \frac{\partial \ell^{(i)}}{\partial \mathbf{h}^{(j)}} \right\| \leq \left\| \frac{\partial \ell^{(i)}}{\partial \mathbf{h}^{(i)}} \right\| \|\mathbf{W}_h\|^{i-j} \prod_{j < t \leq i} \left\| \text{diag} \left( \sigma' \left( \mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b} \right) \right) \right\|$$

If largest singular value of  $\mathbf{W}$  is smaller than 1,  $\left\| \frac{\partial \ell^{(i)}}{\partial \mathbf{h}^{(j)}} \right\|$  shrinks exponentially, assuming sigmoid nonlinearity (gradient  $< 1$ ).



# Improvements on Gradient Descent

- Momentum

- In certain parts of the parameter space, the gradient may oscillate, particularly when ill-conditioned.

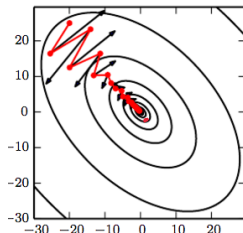


Figure from [2].

- By taking an exponentially decaying weighted sum of past gradients, try to reduce the oscillations and at the same time reinforce gradient in the direction where they all agree.

- Momentum (continued)

- Update equation

$$\underline{\mathbf{v}}^{(t+1)} = \underline{\alpha} \underline{\mathbf{v}}^{(t)} - \underline{\eta} \underline{\nabla_{\mathbf{w}} f(\mathbf{w}^{(t)})},$$

$$\underline{\mathbf{w}}^{(t+1)} = \underline{\mathbf{w}}^{(t)} + \underline{\mathbf{v}}^{(t+1)},$$

where  $\underline{\alpha} \in (0, 1)$  with usual values of 0.5, 0.9 and 0.99.

- $\mathbf{v}$  is called the velocity term

$$\underline{\mathbf{v}}^{(t+1)} = -\underline{\eta} \underline{\nabla_{\mathbf{w}} f(\mathbf{w}^{(t)})} - \underline{\alpha} \underline{\eta} \underline{\nabla_{\mathbf{w}} f(\mathbf{w}^{(t-1)})} - \underline{\alpha^2} \underline{\eta} \underline{\nabla_{\mathbf{w}} f(\mathbf{w}^{(t-2)})} - \dots,$$

Opposing components tend to cancel, and components that agree reinforce each other.

- Nesterov momentum uses the current velocity to update the weight before computing the gradient – for smooth convex batch case, this accelerates convergence.

$$\underline{\mathbf{v}}^{(t+1)} = \underline{\alpha} \underline{\mathbf{v}}^{(t)} - \underline{\eta} \underline{\nabla_{\mathbf{w}} f(\underline{\mathbf{w}}^{(t)} + \underline{\alpha} \underline{\mathbf{v}}^{(t)})},$$

$$\underline{\mathbf{w}}^{(t+1)} = \underline{\mathbf{w}}^{(t)} + \underline{\mathbf{v}}^{(t+1)},$$



- Component-wise adaptation. Some algorithms adapt the learning rates individually for each weight.
  - Adagrad scales the learning rate inversely proportional to the square root of the sum of squared historical gradient values

$$\underline{\mathbf{r}^{(t+1)}} = \underline{\mathbf{r}^{(t)}} + \underline{\mathbf{g} \odot \mathbf{g}},$$

$$\underline{\mathbf{w}^{(t+1)}} = \underline{\mathbf{w}^{(t)}} - \frac{\eta}{\underline{\delta + \sqrt{\mathbf{r}^{(t+1)}}}} \odot \underline{\mathbf{g}},$$

where  $\delta$  is a small constant, and  $\odot$  denotes component-wise multiplication (and division for  $\mathbf{r}$ ).

$$\begin{bmatrix} g_1 \\ \vdots \\ g_n \end{bmatrix} \odot \begin{bmatrix} g_1 \\ \vdots \\ g_n \end{bmatrix} = \begin{bmatrix} g_1^2 \\ \vdots \\ g_n^2 \end{bmatrix}$$

- Component-wise adaptation (continued)
  - Adagrad is designed for convex functions. RMSprop modifies the sum of squared gradient values to a exponentiated weighted sum to reduce the effects of earlier values.

$$\mathbf{r}^{(t+1)} = \rho \mathbf{r}^{(t)} + \underbrace{(1 - \rho)}_{\text{decay}} \mathbf{g} \odot \mathbf{g},$$

$$\underbrace{\mathbf{w}}^{(t+1)} = \mathbf{w}^{(t)} - \frac{\eta}{\underbrace{\delta + \sqrt{\mathbf{r}^{(t+1)}}}_{\text{adaptive step size}}} \odot \mathbf{g}.$$

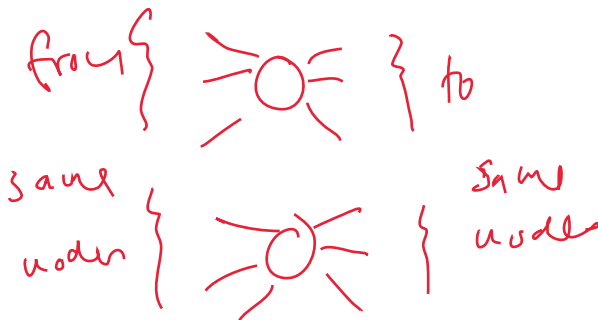
- Combining component-wise adaptation with momentum
  - Adam rescale the velocity vector (from momentum) rather than the gradient. It also increases the estimate of  $\mathbf{v}$  and  $\mathbf{r}$  at the start of training.

$$\begin{aligned}
 \underline{\mathbf{v}}^{(t+1)} &= \rho_1 \underline{\mathbf{v}}^{(t)} + (1 - \rho_1) \underline{\mathbf{g}}, \\
 \underline{\mathbf{r}}^{(t+1)} &= \rho_2 \underline{\mathbf{r}}^{(t)} + (1 - \rho_2) \underline{\mathbf{g}} \odot \underline{\mathbf{g}}, \\
 \underline{\hat{\mathbf{v}}} &= \underline{\mathbf{v}}^{(t+1)} / (1 - \rho_1^{t+1}); \quad \underline{\hat{\mathbf{r}}} = \underline{\mathbf{r}}^{(t+1)} / (1 - \rho_2^{t+1}), \\
 \underline{\mathbf{w}}^{(t+1)} &= \underline{\mathbf{w}}^{(t)} - \frac{\eta}{\delta + \sqrt{\underline{\hat{\mathbf{r}}}}} \odot \underline{\hat{\mathbf{v}}}.
 \end{aligned}$$

- Approximate second order methods
  - Not possible to apply Newton's method for large networks as Hessian size is squared the number of parameters.
  - Approximate methods are possible. Some of them include
    - Conjugate gradients
    - Limited memory BFGS

# Initialization

- Initialization can determine whether the algorithm converges to a good local minimum, also whether it converges quickly.
- Need to break symmetry. If two hidden units have identical initial parameters and connections, a deterministic algorithm will keep them the same throughout.



- Normally initialize using weights drawn from Gaussian or uniform distribution, usually centered around zero, standard deviation or range of size around  $1/\sqrt{d}$ , where  $d$  is the input dimension of the unit.



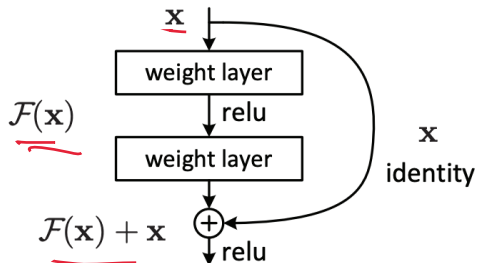
## Demo (Credit: Alec Radford <https://twitter.com/alecrad>)

- Overshooting: [http://sebastianruder.com/content/images/2016/09/contours\\_evaluation\\_optimizers.gif](http://sebastianruder.com/content/images/2016/09/contours_evaluation_optimizers.gif)
  - SGD slow, momentum and Nesterov overshoot, others head towards the right direction quickly.
- Saddle point:  
[http://sebastianruder.com/content/images/2016/09/saddle\\_point\\_evaluation\\_optimizers.gif](http://sebastianruder.com/content/images/2016/09/saddle_point_evaluation_optimizers.gif)
  - SGD gets stuck, Momentum and Nesterov slow but eventually escapes, others okay.

# Improving Optimization by Architectural Design

- Overparameterization, using more parameters than necessary, appears to make gradient based optimization easier.
- ReLU activation appears to be better for optimization.
- Gating architectures, like LSTM, tries to send gradient unchanged by approximately leaving value of state unchanged, unless gating unit allows change (previously discussed).
- Residual neural network (ResNet) adds skip connections between layers, allowing most paths to the output to be short.
- Batch normalization adds parameters for normalizing units in each hidden layer – normalized inputs usually have better optimization properties,

- ResNet

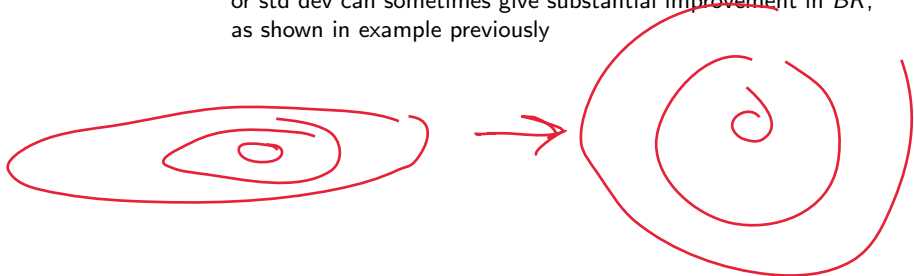


Skip connection. Figure from [3].

- ResNet blocks include skip connections. Output of block is sum of previous layer and skipped block ( $\mathcal{F}$  in figure is single hidden layer network).
- If  $\mathcal{F}$  set to zero function, become shallower network.
- In deep networks, skipped connections provide many shorter connections to earlier layers.

- Batch normalization

- Recall for learning linear function with convex loss, (stochastic) gradient descent has expected excess loss bounded by  $\frac{\rho BR}{\sqrt{T}}$  where  $\|\mathbf{w}^*\|_2 \leq \underline{B}$ ,  $\|\mathbf{x}\|_2 \leq \underline{R}$ , and  $\rho$  is the Lipschitz constant.
  - Normalizing each component of the input to have unit range or std dev can sometimes give substantial improvement in  $BR$ , as shown in example previously



- What about for deep networks? Batch normalization tries to apply the same intuition.

- ... (continued)
  - Batch normalization procedure (computed for each mini-batch with  $m$  examples): At each layer

$$\mu_{\zeta} = \frac{1}{m} \sum_{i=1}^m x_i$$

$$\sigma_{\zeta}^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\zeta})^2$$

$$\hat{x}_i = \frac{x_{\zeta} - \mu_{\zeta}}{\sqrt{\sigma_{\zeta}^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

- At inference time, estimate  $\mu_{\zeta}$  and  $\sigma_{\zeta}$  from entire training set instead of mini-batch.

- ... continued
  - Batch normalization normally applied before nonlinear activation function
  - To retain the capability to represent original function, two parameters: scale  $\gamma$  and shift parameters  $\beta$  added to each unit, transforming it to  $y_i$

$$y_i = \gamma \hat{x}_i + \beta \quad \text{learnable}$$

To get back  $y_i = x_i$

$$\text{set } \gamma = \frac{1}{\sqrt{\sigma^2 + \epsilon}}$$

$$\beta = \mu_i$$

- Backpropagation can still be done to compute gradient after batch normalization
  - Gradient propagated through the transformation (whole batch becomes a single network).

- ... continued
  - Following intuition of linear function with convex loss, we hope batch normalization would speed up learning for next layer.
    - Mean zero and std dev of 1 for the inputs to the next layer (with changes made by  $\gamma$ ,  $\beta$  and nonlinear activation)
    - Works well in practice in speeding up learning, allowing larger learning rate.
  - Reduces change in statistics (the mean and std dev) of the unit, the covariate shift, to the next layer as a result of change in earlier layers.

# Outline

- 1 Deep Learning
- 2 Boosting and Decision Trees
- 3 EM/K-means
- 4 Appendix



# Boosting

Boosting started as a theoretical question of whether efficient weak learning implies efficient strong (PAC) learning.

**Definition (Weak Learning):** For  $\gamma > 0$ , an algorithm  $A$  is called a  $\gamma$ -weak learning algorithm for a Boolean function  $f$  if for any distribution  $\mathcal{D}$  over  $\mathcal{X}$ , the algorithm takes a set of examples from distribution  $\mathcal{D}$  as input and outputs a hypothesis  $h \in \mathcal{H}$  with error at most  $1/2 - \gamma$ , i.e.  $\Pr_{\mathcal{D}}(h(x) \neq f(x)) \leq 1/2 - \gamma$ .

A weak learner is guaranteed to find a hypothesis that is slightly better than chance. Does efficient weak learning imply efficient PAC learning?

# AdaBoost

- AdaBoost proceeds in rounds, where in each round the weak learner adds a classifier to a linear combination of classifiers.
- Let  $S = (\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)$  be the training examples.
- At round  $t$ , AdaBoost defines a distribution  $\mathbf{D}^{(t)}$  over  $S$ .
- The weak learner returns a hypothesis  $h_t$  whose error with respect to  $\mathbf{D}^{(t)}$  is

$$\epsilon_t = L_{\mathbf{D}^{(t)}}(h_t) = \sum_{i=1}^m D_i^{(t)} \mathbb{1}_{[h_t(\mathbf{x}_i) \neq y_i]}.$$

- A weight  $w_t = \frac{1}{2} \log \left( \frac{1}{\epsilon_t} - 1 \right)$  is assigned to  $h_t$ .
- The distribution  $\mathbf{D}^{(t)}$  is updated by multiplying  $D_i^{(t)}$  with  $\exp(-w_t y_i h_t(\mathbf{x}_i))$  and renormalizing to make it a new probability distribution  $\mathbf{D}^{(t+1)}$ .
  - If the classification is correct  $y_i h_t(\mathbf{x}_i) = 1$  and the example is downweighted as  $\exp(-w_t y_i h_t(\mathbf{x}_i)) < 1$ , otherwise it is upweighted.



- AdaBoost focuses attention of the weak learner on examples that most previous classifiers get wrong.

### AdaBoost

input:

training set  $S = (\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)$

weak learner WL

number of rounds  $T$

initialize  $\mathbf{D}^{(1)} = (\frac{1}{m}, \dots, \frac{1}{m})$ .

for  $t = 1, \dots, T$ :

invoke weak learner  $h_t = \text{WL}(\mathbf{D}^{(t)}, S)$

compute  $\epsilon_t = \sum_{i=1}^m D_i^{(t)} \mathbb{1}_{[y_i \neq h_t(\mathbf{x}_i)]}$

let  $w_t = \frac{1}{2} \log \left( \frac{1}{\epsilon_t} - 1 \right)$

update  $D_i^{(t+1)} = \frac{D_i^{(t)} \exp(-w_t y_i h_t(\mathbf{x}_i))}{\sum_{j=1}^m D_j^{(t)} \exp(-w_t y_j h_t(\mathbf{x}_j))}$  for all  $i = 1, \dots, m$

output the hypothesis  $h_s(\mathbf{x}) = \text{sign} \left( \sum_{t=1}^T w_t \text{ $h_t(\mathbf{x})$  \right)$ .

Figure from SSBD.

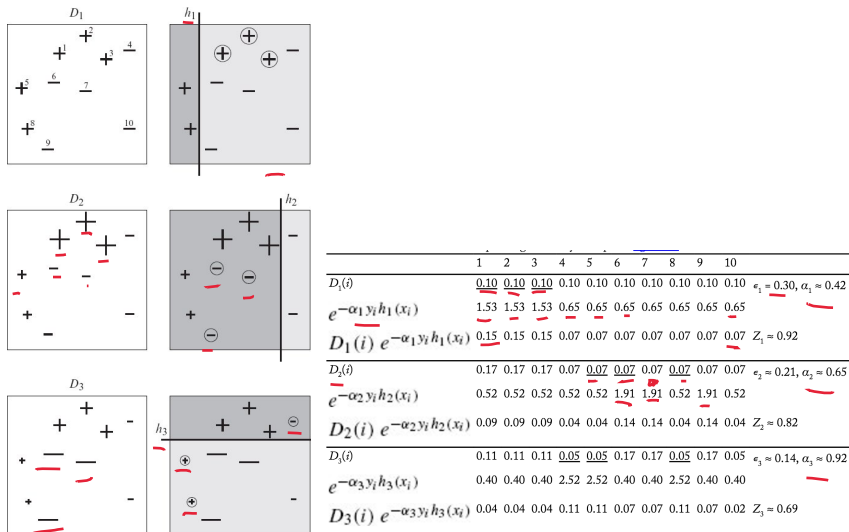


Figure: Example from [6]. Three rounds of AdaBoost.

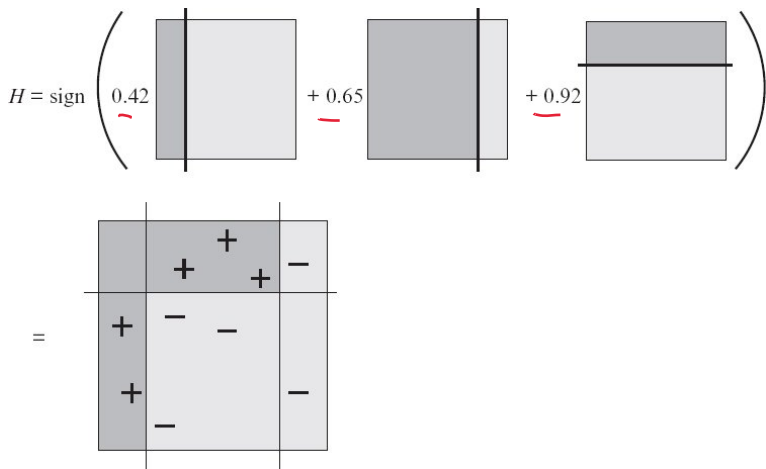


Figure: Example from [6]. Resulting hypothesis.

**Theorem:** (SSBD Theorem 10.2) Let  $S$  be a training set and assume that at each iteration of AdaBoost, the weak learner returns a hypothesis for which  $\epsilon_t \leq 1/2 - \gamma$ . Then, the training error of the output hypothesis of AdaBoost is at most

$$L_S(h_t) = \frac{1}{m} \sum_{i=1}^m \mathbb{1}_{[h_t(\mathbf{x}_i) \neq y_i]} \leq \exp(-2\gamma^2 T).$$

Proof in Appendix.

# Sample Complexity

- The classification error is bounded by  $\frac{1}{m} \sum_{i=1}^m \mathbb{1}_{[h_s(\mathbf{x}_i) \neq y_i]} \leq \exp(-2\gamma^2 T)$ .
- As errors must be multiples of  $1/m$ , setting  $\exp(-2\gamma^2 T) = 1/m$  shows that  $T = \frac{\log m}{2\gamma^2}$  is sufficient to classify all training examples correctly. —



**Lemma:** (SSBD Lemma 10.3) Let  $B$  be a base class and let  $L(B, T)$  be a thresholded linear combination of  $T$  functions from  $B$ . Assume that both  $T$  and  $VCdim(B)$  are at least 3. Then,

$$VCdim(L(B, T)) \leq T(VCdim(B)+1)(3\log(T(VCdim(B)+1))+2).$$

The proof is essentially the same as the proof of for the  $VCdim$  of neural networks.

$$T = \tilde{O}\left(\frac{\log n}{2\epsilon^2}\right)$$

$$VCdim = \tilde{O}\left(\frac{\log n}{2\epsilon^2} VCdim(B)\right)$$

$\tilde{O}$  is  $\overline{O}$ -notation that ignores  $\log$  factors.

- Can also show that margin keeps increasing as we keep running AdaBoost after training error is zero.
  - In practice, generalization error often keeps decreasing.
  - Also have margin-based generalization bound that shows that generalization improves as margin improves.

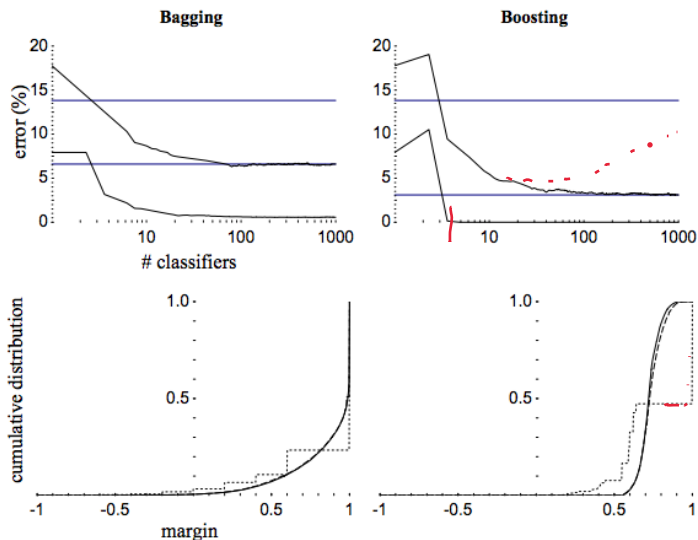


Figure from [7]. Boosting and bagging decision trees on letter dataset.

## Exercise O2-2 (Archipelago)

Let  $\epsilon_t$  be the error of the weak learner  $h_t$  in round  $t$ , on distribution  $D^{(t)}$ . The distribution  $D^{(t)}$  is updated to  $D^{(t+1)}$  as follows then reweighted:

- if  $h_t$  misclassifies  $\mathbf{x}$  (with probability  $\epsilon_t$  under  $D^{(t)}$ ),  $D^{(t)}(\mathbf{x})$  is multiplied by  $\exp(w_t)$
- otherwise (with probability  $1 - \epsilon_t$  under  $D^{(t)}$ ), it is multiplied by  $\exp(-w_t)$

where  $w_t = \ln \frac{\sqrt{1-\epsilon_t}}{\sqrt{\epsilon_t}}$ .

$$e^{w_t} = \frac{\sqrt{1-\epsilon_t}}{\sqrt{\epsilon_t}} \quad |$$

$$e^{-w_t} = \frac{\sqrt{\epsilon_t}}{\sqrt{1-\epsilon_t}}$$

What is the value of  $\Pr_{D^{(t+1)}}(h_t(x) \neq y_t)$ ?

*Advantage of  $h_t$  disappears under  $D^{(t+1)}$*

$$\begin{aligned}
 \sum_{i: y_i \neq h_t(x_i)} D_i^{(t+1)} &= e^{-w_t} \sum_{i: y_i \neq h_t(x_i)} D_i^{(t)} \\
 &= \frac{\sqrt{(1-\epsilon_t)} \epsilon_t}{\sqrt{\epsilon_t}} = \sqrt{(1-\epsilon_t)} \sqrt{\epsilon_t}
 \end{aligned}$$

*same*

$$\begin{aligned}
 \sum_{i: y_i = h_t(x_i)} D_i^{(t+1)} &= e^{-w_t} \sum_{i: y_i = h_t(x_i)} D_i^{(t)} \\
 &= \frac{\sqrt{\epsilon_t} (1-\epsilon_t)}{\sqrt{(1-\epsilon_t)}} = \sqrt{(1-\epsilon_t)} \sqrt{\epsilon_t}
 \end{aligned}$$

### Exercise O2-3 (Archipelago)

$$\frac{(1 - \epsilon_t)}{\epsilon_t}$$

Consider the weight  $w_t = \frac{1}{2} \ln \frac{1 - \epsilon_t}{\epsilon_t}$  of  $h_t$ . What is its behaviour when:

- 1  $\epsilon > 1/2$   $w_t < 0$   $-h_t$  is good
- 2  $\epsilon < 1/2$   $w_t > 0$  weak learner useful
- 3  $\epsilon = 1/2$   $w_t = 0$

Does the behaviour make intuitive sense?

# Decision Trees

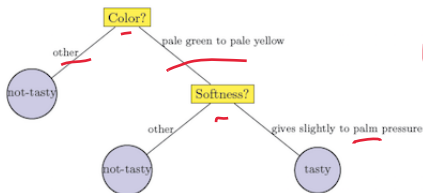
- Decision trees work well in combination with boosting.
  - Training a single decision tree has advantages in interpretability if the resulting tree is small.
  - But boosted (or bagged) trees have higher accuracy.
- We consider only binary features in the description.

- Previously we used minimum description length to get

$$L_{\mathcal{D}}(h) \leq L_S(h) + \sqrt{\frac{(n+1) \log_2(d+3) + \log(2/\delta)}{2m}},$$

where  $n$  is the number of node in the tree.

- But minimizing the bound is NP-hard. Instead we use a greedy algorithm.
  - A decision tree recursively partitions the space into non-overlapping axis-aligned rectangles.
  - Learning algorithm similarly recursively partitions the training set using one of the features.
  - Feature used to partition usually selected by greedily optimizing a gain measure.



Other  
not  
Tasty

color  
pale green ... NT  
own softness  
give slightly  
T

Figure from SSBD.



# ID3(S, A)

INPUT: training set S, feature subset A  $\subseteq [d]$

if all examples in S are labeled by 1, return a leaf 1

if all examples in S are labeled by 0, return a leaf 0

if A =  $\emptyset$ , return a leaf whose value = majority of labels in S

else :

Let  $j = \operatorname{argmax}_{i \in A} \text{Gain}(S, i)$

if all examples in S have the same label

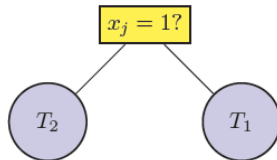
Return a leaf whose value = majority of labels in S

else

Let T<sub>1</sub> be the tree returned by ID3( $\{(x, y) \in S : x_j = 1\}, A \setminus \{j\}$ ).

Let T<sub>2</sub> be the tree returned by ID3( $\{(x, y) \in S : x_j = 0\}, A \setminus \{j\}$ ).

Return the tree:

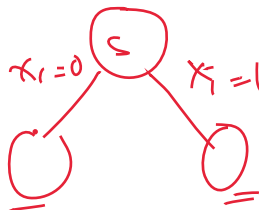


not needed

Pseudo-code for ID3 algorithm. Figure from SSBD.

- Define an impurity measure on a node as a function of the probability of the possible labels at the node. For binary classification, we have impurity measure  $C(P_S[y = 1])$  where  $P_S[y = 1]$  is the probability of label 1 in the partition  $S$  at that node.
- The gain measure  $G(S, i)$  is

$$G(S, i) = C(P_S[y = 1]) - (P_S[x_i = 1]C(P_S[y = 1|x_i = 1]) + P_S[x_i = 0]C(P_S[y = 1|x_i = 0])).$$



- Commonly used purity measure includes
  - Entropy, resulting in information gain measure:

$$C(a) = -a \log a - (1 - a) \log(1 - a).$$

ID3

- Gini impurity, resulting in the Gini index measure:

$$C(a) = 2a(1 - a).$$

CART



Which attribute is the best classifier?

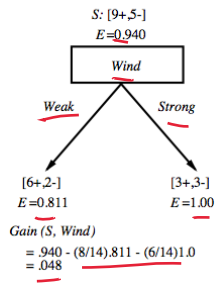
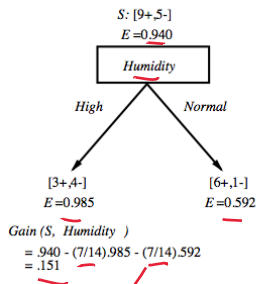
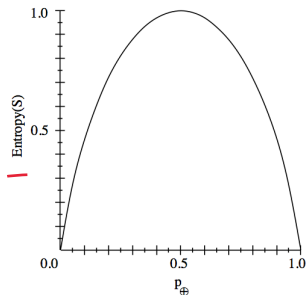
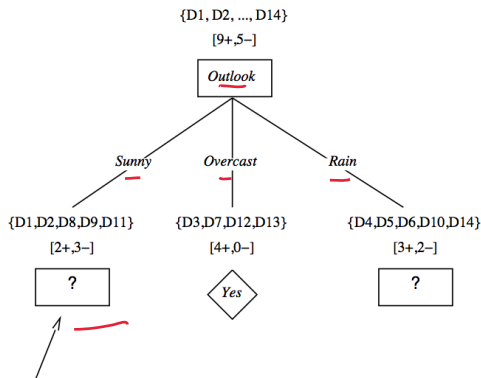


Figure: From [4]. Entropy function on the left. Notice that entropy is small when probability of positive example is close to 0 or 1. On the right, information gain computation for two features.



Which attribute should be tested here?

$$S_{\text{sunny}} = \{D1, D2, D8, D9, D11\}$$

$$\text{Gain}(S_{\text{sunny}}, \text{Humidity}) = .970 - (3/5) 0.0 - (2/5) 0.0 = .970$$

$$\text{Gain}(S_{\text{sunny}}, \text{Temperature}) = .970 - (2/5) 0.0 - (2/5) 1.0 - (1/5) 0.0 = .570$$

$$\text{Gain}(S_{\text{sunny}}, \text{Wind}) = .970 - (2/5) 1.0 - (3/5) .918 = .019$$

**Figure:** From [4]. Partially learned decision tree. Descendent of *Overcast* have only positive labels, so becomes a leaf node. The other two nodes will be expanded using the feature with highest information gain.

After growing the tree, we often prune it to minimize an estimate of the generalization error, e.g. the MDL bound, or validation set error (often called reduced error pruning).

## Generic Tree Pruning Procedure

input:

function  $\underline{f(T, m)}$  (bound/estimate for the generalization error of a decision tree  $T$ , based on a sample of size  $m$ ), tree  $T$ .

foreach node  $j$  in a bottom-up walk on  $T$  (from leaves to root):

find  $T'$  which minimizes  $f(T', m)$ , where  $T'$  is any of the following:

- the current tree after replacing node  $j$  with a leaf 1.
- the current tree after replacing node  $j$  with a leaf 0.
- the current tree after replacing node  $j$  with its left subtree.
- the current tree after replacing node  $j$  with its right subtree.
- the current tree.

let  $T := T'$ .

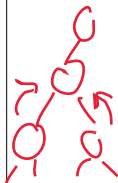


Figure from SSBD.

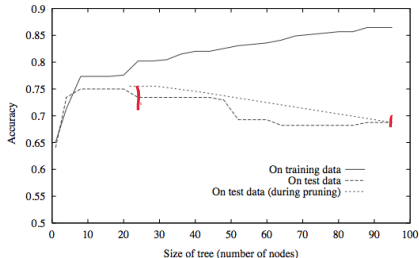
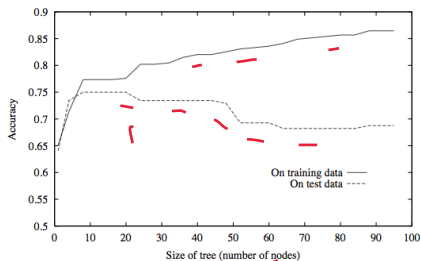


Figure: From [4]. Left figure shows training and test error on a diabetes dataset. Right figure include the test error when pruning is done.

## Exercise O2-4 (Archipelago)

Consider the following training set, where  $X = \{0, 1\}^3$  and  $Y = \{0, 1\}$ :

*Handwritten notes:  $f_1, f_2, f_3$  and entropy = 1*

$x_1$	$((1, \underline{1}, \underline{1}), \underline{1})$
$x_2$	$((1, \underline{0}, \underline{0}), 1)$
$x_3$	$((\underline{1}, \underline{1}, \underline{0}), \underline{0})$
$x_4$	$((\underline{0}, \underline{0}, \underline{1}), \underline{0})$

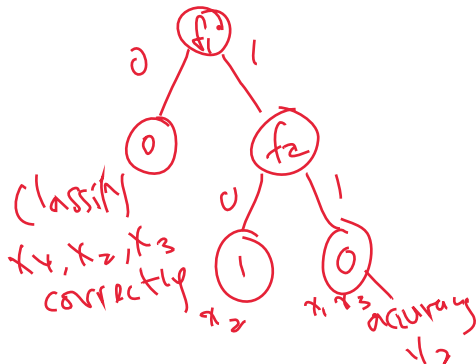
*Handwritten note: no change in entropy*

Suppose we wish to use this training set in order to build a decision tree of **depth 2** (i.e., for each input we are allowed to ask two questions of the form  $(x_i = 0?)$  before deciding on the label).

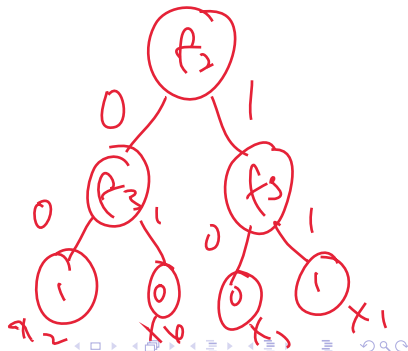


What is ID3's error rate when it is run to depth 2? What is the optimal error rate of a depth 2 decision tree?

First split is  $f_1$



Opt Depth 2 tree



# AdaBoost as Coordinate Descent

- Let  $\underline{S} = ((\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m))$ .
- Let  $\underline{L_S}(f) = \frac{1}{m} \sum_{i=1}^m \ell(f(\underline{x_i}), y_i)$  denote the empirical risk.
- $\underline{L_S}(f) = \frac{1}{m} \sum_{i=1}^m e^{-y_i \underline{f(\mathbf{x_i})}}$  is the empirical risk used in AdaBoost.
  - We have used  $e^{-y \underline{f(\mathbf{x})}}$  as the surrogate loss, and  $f(\mathbf{x}) = \sum_t w_t h_t(\mathbf{x})$ .
- Consider a finite space of base hypotheses  $\mathcal{H} = \{h_1, \dots, h_N\}$ .
- Can write  $L_S(w_1, \dots, w_N) = \frac{1}{m} \sum_{i=1}^m e^{-y_i \sum_{j=1}^N \underline{w_j h_j(\mathbf{x_i})}}$ .

- In coordinate descent, at each iteration, we want to find  $j, \alpha$  to minimize  $L_S(w_1, \dots, w_{j-1}, w_j + \alpha, w_{j+1}, \dots, w_N)$ .
- Can show that AdaBoost exactly selects  $j, \alpha$  for coordinate descent, so can be viewed as a coordinate descent algorithm for the exponential loss.

AdaBoost selects same  $j$   
 and same  $\alpha \rightarrow \ln \frac{1-\epsilon}{\epsilon}$   
 as coord descent.

# Functional Gradient Descent

- Coordinate descent works out very well for the exponential loss.
  - Find the base classifier that minimizes a reweighted classification error.
  - Closed form weight update.
- But need not be true for general loss functions.
- Functional gradient descent provides a different generalization that is advantageous in some cases.
- In functional gradient descent, the objective function takes a function as input, e.g. in AdaBoost

$$\mathcal{L}_S(F) = \frac{1}{m} \sum_{i=1}^m e^{-y_i F(\mathbf{x}_i)}.$$

$\mathcal{L}_S$  is called a functional (maps a function to a real number).

- In this case, we are interested in the value of  $F$  at  $m$  points  $\mathbf{x}_1, \dots, \mathbf{x}_m$ .
  - The output of the function is a vector of length  $m$ ,  $(f_1, \dots, f_m) = (F(\mathbf{x}_1), \dots, F(\mathbf{x}_m))$  and  $\mathcal{L}(F)$  can be thought as a function on  $\mathbb{R}^m$ .
- As  $\mathcal{L}_S(F) = \frac{1}{m} \sum_{i=1}^m \ell(f_i, y_i)$ , the gradient is a  $m$  dimensional vector corresponding to the  $m$  examples. The  $i$ -th component of the gradient is

$$\nabla \mathcal{L}_S(F)_i = \frac{1}{m} \frac{\partial \ell(F(\mathbf{x}_i), y_i)}{\partial F(\mathbf{x}_i)}.$$

- But we cannot necessarily update in direction  $\nabla \mathcal{L}_S(F)$ , as we are restricted to directions  $(h(\mathbf{x}_1), \dots, h(\mathbf{x}_m))$  for  $h \in \mathcal{H}$ .
- Consider a first order Taylor series approximation of  $\mathcal{L}_S(F)$  in direction  $h$

$$\mathcal{L}_S(F + wh) \approx \mathcal{L}_S(F) + w \langle \nabla \mathcal{L}_S(F), h \rangle.$$

- At each step of boosting, we try to find  $h$  that minimizes this approximation, or equivalently maximizes  $-\langle \nabla \mathcal{L}_S(F), h \rangle$

- After  $t$  rounds of boosting, the value of  $F$  is  $f_t$  where  $f_t(\mathbf{x}) = \sum_{p=1}^t w_p h_p(\mathbf{x})$  and the gradient is  $\nabla_t \mathcal{L}_S(f_t)$ .
  - For AdaBoost  $\nabla_t \mathcal{L}_S(f_t)(\mathbf{x}_i) = -y_i \exp(-y_i f_t(\mathbf{x}_i))/m$ , so

$$-\langle \nabla_t \mathcal{L}_S(f), h \rangle = \sum_{i=1}^m y_i h(\mathbf{x}_i) \exp(-y_i f_t(\mathbf{x}_i))/m.$$

- After normalization, this gives the interpretation of finding  $h$  that minimizes the error on distribution  $D^{(t)}$ .
- After finding  $h$ , we want to update  $f_{t+1} = f_t + wh$ .
- The AdaBoost update corresponds to the value  $w$  that optimizes the empirical risk  $L_S(f) = \frac{1}{m} \sum_{i=1}^m e^{-y_i(f_t(\mathbf{x}_i) + wh(\mathbf{x}_i))}$ .

- Instead of using the exponential loss, can use other loss functions such as squared loss, logistic loss. Can also add regularization term to minimize regularized loss.
  - For the exponential loss, maximizing  $-\langle \nabla_t \mathcal{L}_S(f), h \rangle$  gives the same  $h$  as in coordinate descent.
  - For other loss function, the function  $h$  that is selected may be different from that selected for coordinate descent.
- This gives the gradient boosting algorithm.
- Commonly used with trees as weak learners.



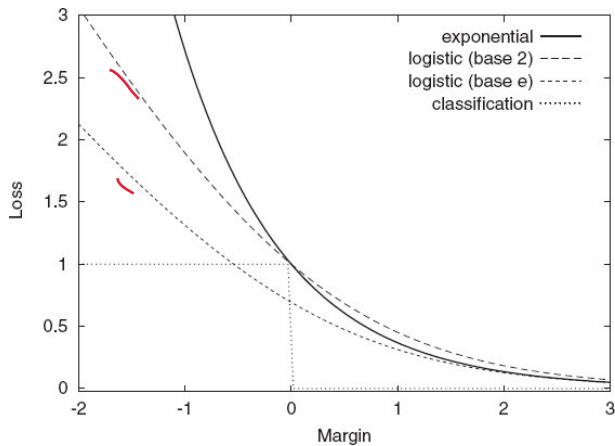


Figure: Figure from [7]. Exponential and logistic losses.

## XGBoost

- Favourite method in Kaggle competitions.
  - Among 29 winning solutions in 2015, 17 used XGBoost.
  - Deep neural networks second most popular, used in 11 solutions.
- Used by every winning team in top 10 of KDDCup 2015.
- Won the 2016 John Chambers Statistical Software Award.
- Tree ensemble model.
- Like gradient boosting, but use second order approximation (instead of just gradient) and regularization.
- Also select random variable subset to consider for splitting, like random forest.

# Outline

- 1 Deep Learning
- 2 Boosting and Decision Trees
- 3 EM/K-means**
- 4 Appendix

# Mixture Model

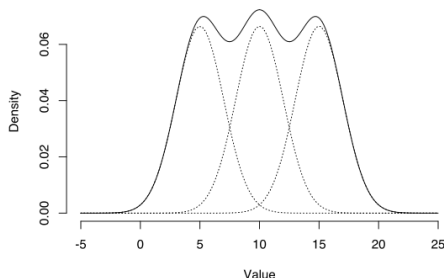
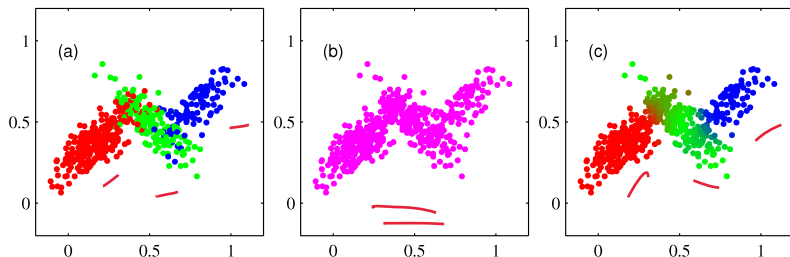


Figure: From SSBD. Mixture of Gaussians

- We assume that each data point  $\mathbf{x}$  is generated by randomly selecting one of  $k$  Gaussians using  $P[Y = k] = \underline{c_k}$  and then generating  $\mathbf{x}$  using the distribution with mean  $\mu_k$  and covariance matrix  $\Sigma_k$ .



**Figure:** From [1]. Mixture of 3 Gaussians. Left figure shown in red, green, blue corresponding to the three mixtures. Middle is sample from marginal  $p(\mathbf{x})$ . Right shows the estimated component for each point using proportions of red, blue and green.

- The marginal distribution is

$$\begin{aligned}
 P(X = \mathbf{x}) &= \sum_{y=1}^k P(Y = y) P(X = \mathbf{x} | Y = y) \\
 &= \sum_{y=1}^k c_y \frac{1}{(2\pi)^{d/2} |\Sigma_y|^{1/2}} \exp \left( -\frac{1}{2} (\mathbf{x} - \mu_y)^T \Sigma_y^{-1} (\mathbf{x} - \mu_y) \right).
 \end{aligned}$$

- How do we learn  $c_y$ ,  $\mu_y$  and  $\Sigma_y$  from samples of the marginal?
- A standard method is to maximize the log likelihood of the data, i.e. given i.i.d. sample  $S = (\mathbf{x}_1, \dots, \mathbf{x}_m)$  generated from model with parameter  $\theta$ , maximize

$$\begin{aligned}
 L(\theta) &= \log \prod_{i=1}^m P_{\theta}[X = \mathbf{x}_i] \\
 &= \sum_{i=1}^m \log \left( \sum_{y=1}^k P_{\theta}[X = \mathbf{x}_i, y = y] \right).
 \end{aligned}$$

- Can do gradient ascent (or other optimization algorithm) to maximize the log likelihood.
- Expectation maximization (EM) is a different iterative procedure for finding a local maximum of the log likelihood.
  - Particularly useful for cases where maximum likelihood is easy if the values of the latent variables are known.
  - Can be used beyond mixture models, for other probabilistic models with latent variables such as hidden Markov models.

- We consider the case of discrete  $Y$  taking  $k$  possible values.
- At each step, EM maximizes the **expected log likelihood**

$$F(Q, \theta) = \sum_{i=1}^m \sum_{y=1}^k Q_{i,y} \log(P_{\theta}[X = x_i, Y = y]),$$

where  $Q_{i,y}$  forms a probability distribution over  $y$ .

- The steps of the EM algorithm are to repeatedly do the following

- 1 Expectation step: Set

$$Q_{i,j}^{(t+1)} = P_{\theta^{(t)}}[Y = y_j | X = x_i].$$

- 2 Maximization step: Set  $\theta^{(t+1)}$  to the maximizer of the expected log likelihood, where the expectation is defined by  $Q^{(t+1)}$

$$\theta^{(t+1)} = \arg \max_{\theta} F(Q^{(t+1)}, \theta).$$



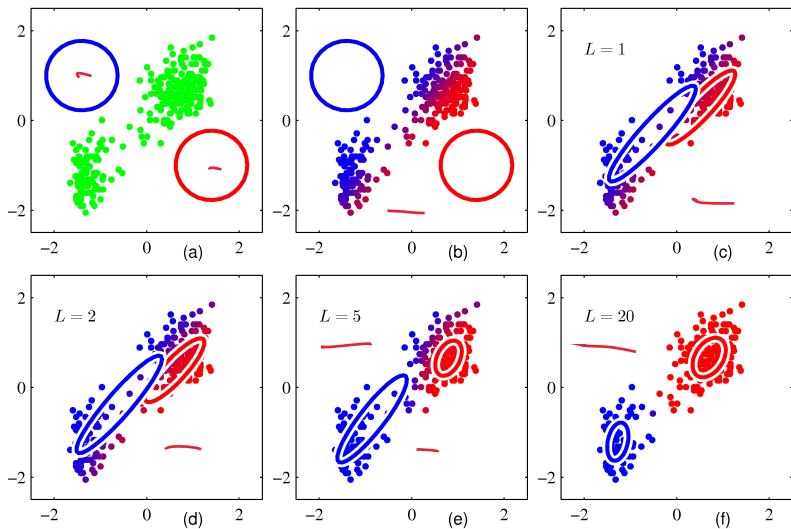


Figure: From [1]. Illustration of EM algorithm on Old Faithful dataset.

# EM as Alternate Maximization Algorithm

We can view EM as an alternate maximization algorithm for the following objective

$$\mathcal{L}(Q, \theta) = F(Q, \theta) - \sum_{i=1}^m \sum_{y=1}^k Q_{i,y} \log(Q_{i,y}).$$

*Handwritten notes:*  
 Evidence Q  
 Lower bound  
 Expected log likelihood  
 ELBO =  $\sum_y g(y) \log \frac{P(x,y)}{g(y)}$   
 $= \sum_y g(y) \log \frac{P(x)P(y|x)}{g(y)}$   
 Entropy  $= \sum_y g(y) \log P(x)$

The second term is the sum of entropies over the examples.

**Lemma:** (SSBD Lemma 24.2) The EM procedure can be rewritten as

$$Q^{(t+1)} = \arg \max_Q \mathcal{L}(Q, \theta^{(t)}).$$

$$\theta^{(t+1)} = \arg \max_{\theta} \mathcal{L}(Q^{(t+1)}, \theta).$$

$= \sum_y g(y) \log \frac{g(y)}{P(y|x)}$   
 $= \log P(x)$   
 $- KL(g || p(y|x))$

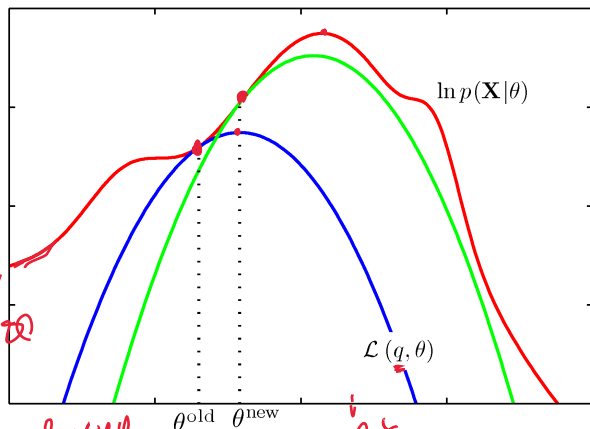
Furthermore

$$\mathcal{L}(Q^{(t+1)}, \theta^{(t)}) = L(\theta^{(t)}).$$

- As a corollary, EM never decreases the log likelihood, as

$$\underline{L(\theta^{(t+1)})} = \underline{\mathcal{L}(Q^{(t+2)}, \theta^{(t+1)})} \geq \underline{\mathcal{L}(Q^{(t+1)}, \theta^{(t)})} = \underline{L(\theta^{(t)})}.$$

E-step  $q(y|x)$   
 is sometimes  
 intractable  
 to compute.  
 Often use  
 approx distr  
 $q(y|x)$   
 s.t. easier  
 to ~~optimize~~ optimize  
 $KL(q||p)$



Variational inference  
 becomes variational  
 EM

Figure: From [1].

- We now consider the EM algorithm for mixture of Gaussians
  - $\theta$  is triplet  $(\underline{\mathbf{c}}, \{\underline{\mu}_1, \dots, \underline{\mu}_k\}, \{\underline{\Sigma}_1, \dots, \underline{\Sigma}_k\})$ , where  $P[Y = k] = \underline{c}_k$ , the means are  $\underline{\mu}_k$  and covariance matrices are  $\underline{\Sigma}_k$ .
  - For simplicity, assume  $\underline{\Sigma}_1 = \underline{\Sigma}_2, \dots, \underline{\Sigma}_k = I$ , the identity matrix.

- Expectation step:

$$\begin{aligned}
 P_{\theta^{(t)}}[Y = y | X = \mathbf{x}_i] &= \frac{1}{Z_i} P_{\theta^{(t)}}[Y = y] P_{\theta^{(t)}}[X = \mathbf{x}_i | Y = y] \\
 &= \frac{1}{Z_i} c_y^{(t)} \exp\left(-\frac{1}{2} \|\mathbf{x}_i - \boldsymbol{\mu}_y^{(t)}\|^2\right),
 \end{aligned}$$

where  $Z_i$  is a normalization constant.

- Maximization step: the expected log likelihood is

$$\sum_{i=1}^m \sum_{y=1}^k P_{\theta^{(t)}}[Y = y | X = \mathbf{x}_i] \left( \log(c_y) - \frac{1}{2} \|\mathbf{x}_i - \boldsymbol{\mu}_y\|^2 \right).$$

Differentiating we get a weighted sum of  $\mathbf{x}_i$  for  $\boldsymbol{\mu}_y$

$$\boldsymbol{\mu}_y = \frac{\sum_{i=1}^m P_{\theta^{(t)}}[Y = 1 | X = \mathbf{x}_i] \mathbf{x}_i}{\sum_{i=1}^m P_{\theta^{(t)}}[Y = y | X = \mathbf{x}_i]}.$$

- Continued...

For  $\mathbf{c}$  we need to do constrained optimization to ensure that it sums to 1. Using the Lagrange multiplier method, we need to maximize

$$\sum_{i=1}^m \sum_{y=1}^k P_{\theta^{(t)}}[Y = y | X = \mathbf{x}_i] \log(c_y) - \lambda \left( \sum_{y=1}^k c_y - 1 \right).$$

Differentiating with respect to  $\lambda$  and setting to zero, we get

$$\sum_{y=1}^k c_y = 1.$$

Differentiating with respect to  $c_y$  and setting the derivative to zero, we get

$$\frac{\sum_{i=1}^m P_{\theta^{(t)}}[Y = y|X = \mathbf{x}_i]}{c_y} = \lambda.$$

Rearranging, we get

$$\sum_{i=1}^m P_{\theta^{(t)}}[Y = y|X = \mathbf{x}_i] = \lambda c_y.$$

Summing both the left and right sides over  $y = 1$  to  $k$ , and using  $\sum_{y=1}^k c_y = 1$ , we get

$$\sum_{i=1}^m \sum_{y=1}^k P_{\theta^{(t)}}[Y = y|X = \mathbf{x}_i] = \lambda \sum_{y=1}^k c_y = \lambda.$$

Substituting  $\lambda$  back, we finally get

$$c_y = \frac{\sum_{i=1}^m P_{\theta^{(t)}}[Y = 1|X = \mathbf{x}_i]}{\sum_{y'=1}^k \sum_{i=1}^m P_{\theta^{(t)}}[Y = y'|X = \mathbf{x}_i]}.$$

# K-means

- K-means is a clustering algorithm for partitioning a set of points in  $\mathbb{R}^d$  into  $k$  clusters.
- Each cluster is represented by the mean vector of the points in the cluster and the aim is to partition the set such that the sum of squared distances of each point to its cluster center is minimized.
- Can be considered as a hard version of the EM algorithm for mixture of Gaussians.



$k$ -Means

**input:**  $\mathcal{X} \subset \mathbb{R}^n$  ; Number of clusters  $k$

**initialize:** Randomly choose initial centroids  $\underline{\mu}_1, \dots, \underline{\mu}_k$

**repeat until convergence**

$\forall i \in [k]$  set  $C_i = \{\underline{\mathbf{x}} \in \mathcal{X} : i = \operatorname{argmin}_j \|\underline{\mathbf{x}} - \underline{\mu}_j\|\}$   
(break ties in some arbitrary manner)

$\forall i \in [k]$  update  $\underline{\mu}_i = \frac{1}{|C_i|} \sum_{\underline{\mathbf{x}} \in C_i} \underline{\mathbf{x}}$

Figure: K-means algorithm, from SSBD

↑  
update each center with mean of cluster

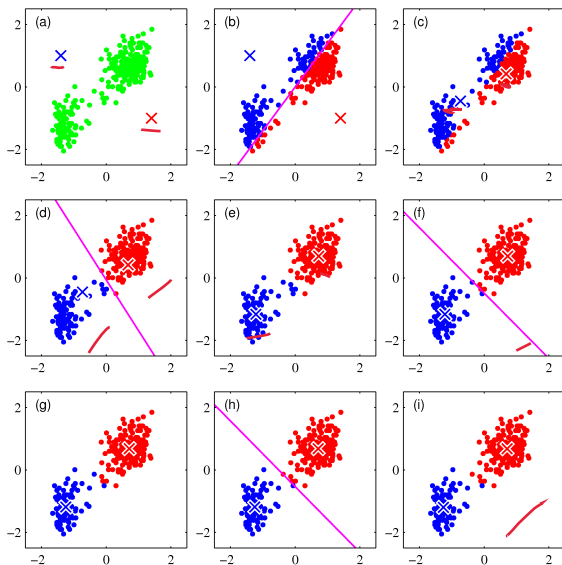
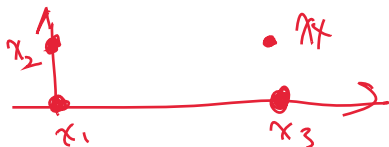


Figure: From [1]. Illustration of  $k$ -means algorithm on Old Faithful dataset.

- Like EM, K-means can be viewed as an alternate optimization algorithm for minimizing the objective

$$\mathcal{L}(\mathbf{C}, \boldsymbol{\mu}) = \sum_{i=1}^k \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2.$$

- The K-means algorithm repeatedly
  - With the centers  $\mu_i$  fixed, assign each point to the cluster that would minimize the distance of that point to the cluster center.
  - With the clusters  $C_i$  fixed, update the centers of each cluster with the mean of the points, which minimizes the sum of squared distances of the points in the cluster to its center.
- This observation gives us the following lemma:  
**Lemma:** (SSBD Lemma 22.1) Each iteration of the k-means algorithm does not increase the k-means objective function.



### Exercise O2-5 (Archipelago)

Consider running K-means with 2 centers on the dataset with 4 points on the plane:  $\mathbf{x}_1 = (0, 0)$ ,  $\mathbf{x}_2 = (0, 2)$ ,  $\mathbf{x}_3 = (2\sqrt{t}, 0)$ , and  $\mathbf{x}_4 = (2\sqrt{t}, 2)$ .

- 1 Consider initializing with  $\mu_1 = \mathbf{x}_1$  and  $\mu_2 = \mathbf{x}_2$ . What is the final sum of squared error? 4t
- 2 Now consider initializing with  $\mu_1 = \mathbf{x}_1$  and  $\mu_2 = \mathbf{x}_3$ . What is the final sum of squared error? 4

Comment on the results.

Can get local optimum  
- arbitrarily bad

# Reading

- 1 SSBD Chapters 10, 22.2.
- 2 Goodfellow, Ian, Yoshua Bengio, and Aaron Courville, “Deep Learning”, <http://www.deeplearningbook.org/>. Chapters 6 and 8.

# References I

- [1] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.
- [3] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.
- [4] Tom M Mitchell. *Machine Learning*. McGraw-Hill Boston, MA: 1997.
- [5] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training recurrent neural networks”. In: *International Conference on Machine Learning*. 2013, pp. 1310–1318.

# References II

- [6] Robert E Schapire and Yoav Freund. *Boosting: Foundations and algorithms*. MIT press, 2012.
- [7] Robert E Schapire et al. “Boosting the margin: A new explanation for the effectiveness of voting methods”. In: *The Annals of Statistics* 26.5 (1998), pp. 1651–1686.

# Outline

- 1 Deep Learning
- 2 Boosting and Decision Trees
- 3 EM/K-means
- 4 Appendix**



# Convergence of AdaBoost

**Theorem:** (SSBD Theorem 10.2) Let  $S$  be a training set and assume that at each iteration of AdaBoost, the weak learner returns a hypothesis for which  $\epsilon_t \leq 1/2 - \gamma$ . Then, the training error of the output hypothesis of AdaBoost is at most

$$L_S(h_T) = \frac{1}{m} \sum_{i=1}^m \mathbb{1}_{[h_S(\mathbf{x}_i) \neq y_i]} \leq \exp(-2\gamma^2 T).$$

*Proof:*

- Denote  $f_t = \sum_{p \leq t} w_p h_p$ . The final output of AdaBoost is  $f_T$ .
- Denote  $Z_t = \frac{1}{m} \sum_{i=1}^m e^{-y_i f_t(\mathbf{x}_i)}$ .
- Note that  $\mathbb{1}_{[h(\mathbf{x}) \neq y]} \leq e^{-yh(\mathbf{x})}$ , i.e. it is a surrogate convex loss function.
- Hence  $L_S(f_T) \leq Z_T$ .

- To upper bound  $Z_T$ , note that  $Z_0 = 1$  because  $f_0 \equiv 0$  and

$$Z_T = \frac{Z_T}{Z_0} = \frac{Z_T}{Z_{T-1}} \frac{Z_{T-1}}{Z_{T-2}} \cdots \frac{Z_2}{Z_1} \frac{Z_1}{Z_0}.$$

- To show  $Z_T \leq e^{-2\gamma^2 T}$ , it suffices to show  $\frac{Z_t}{Z_{t-1}} \leq e^{-2\gamma^2}$  for all  $t$ .
- Note that

$$\begin{aligned} D_i^{(t+1)} &= \frac{D_i^{(t)} \exp(-w_t y_i h_t(\mathbf{x}_i))}{\sum_{j=1}^m D_j^{(t)} \exp(-w_t y_j h_t(\mathbf{x}_j))} \\ &= \frac{D_i^{(t-1)} \exp(-w_t y_i h_{t-1}(\mathbf{x}_i)) \exp(-w_t y_i h_t(\mathbf{x}_i))}{\sum_{j=1}^m D_j^{(t-1)} \exp(-w_t y_j h_{t-1}(\mathbf{x}_j)) \exp(-w_t y_j h_t(\mathbf{x}_j))} \\ &= \cdots = \frac{\exp(-y_i f_t(\mathbf{x}_i))}{\sum_{j=1}^m \exp(-y_j f_t(\mathbf{x}_j))}. \end{aligned}$$

t

- Hence we have

$$\begin{aligned}
 \frac{Z_{t+1}}{Z_t} &= \frac{\sum_{i=1}^m e^{-y_i f_{t+1}(\mathbf{x}_i)}}{\sum_{j=1}^m e^{-y_j f_t(\mathbf{x}_j)}} = \frac{\sum_{i=1}^m e^{-y_i f_t(\mathbf{x}_i)} e^{-y_i w_{t+1} h_{t+1}(\mathbf{x}_i)}}{\sum_{j=1}^m e^{-y_j f_t(\mathbf{x}_j)}} \\
 &= \sum_{i=1}^m D_i^{(t+1)} e^{-y_i w_{t+1} h_{t+1}(\mathbf{x}_i)} \\
 &= e^{-w_{t+1}} \sum_{i: y_i h_{t+1}(\mathbf{x}_i)=1} D_i^{(t+1)} + e^{w_{t+1}} \sum_{i: y_i h_{t+1}(\mathbf{x}_i)=-1} D_i^{(t+1)} \\
 &= e^{-w_{t+1}} (1 - \epsilon_{t+1}) + e^{w_{t+1}} \epsilon_{t+1} \\
 &= \frac{1}{\sqrt{1/\epsilon_{t+1} - 1}} (1 - \epsilon_{t+1}) + (\sqrt{1/\epsilon_{t+1} - 1}) \epsilon_{t+1} \\
 &= \sqrt{\frac{\epsilon_{t+1}}{1 - \epsilon_{t+1}}} (1 - \epsilon_{t+1}) + \sqrt{\frac{1 - \epsilon_{t+1}}{\epsilon_{t+1}}} \epsilon_{t+1} \\
 &= 2\sqrt{\epsilon_{t+1}(1 - \epsilon_{t+1})}.
 \end{aligned}$$

- By assumption,  $\epsilon_{t+1} \leq 1/2 - \gamma$ . As the function  $g(a) = a(1 - a)$  is monotonic in  $[0, 1/2]$  we get

$$2\sqrt{\epsilon_{t+1}(1 - \epsilon_{t+1})} \leq 2\sqrt{\left(\frac{1}{2} - \gamma\right)\left(\frac{1}{2} + \gamma\right)} = \sqrt{1 - 4\gamma^2}.$$

- Using  $1 - a \leq e^{-a}$ , we have  $\sqrt{1 - 4\gamma^2} \leq e^{-4\gamma^2/2} = e^{-2\gamma^2}$ .  $\square$

# Convergence of EM

EM can be viewed as an alternate maximization algorithm for the following objective

$$\mathcal{L}(Q, \theta) = F(Q, \theta) - \sum_{i=1}^m \sum_{y=1}^k Q_{i,y} \log(Q_{i,y}).$$

**Lemma:** (SSBD Lemma 24.2) The EM procedure can be rewritten as

$$Q^{(t+1)} = \arg \max_Q \mathcal{L}(Q, \theta^{(t)}).$$

$$\theta^{(t+1)} = \arg \max_{\theta} \mathcal{L}(Q^{(t+1)}, \theta).$$

Furthermore

$$\mathcal{L}(Q^{(t+1)}, \theta^{(t)}) = L(\theta^{(t)}).$$

*Proof:*

- If  $Q^{(t+1)}$  is fixed, then

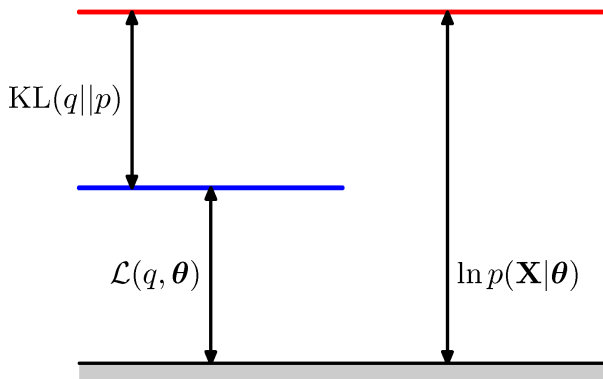
$$\arg \max_{\theta} \mathcal{L}(Q^{(t+1)}, \theta) = \arg \max_{\theta} F(Q^{(t+1)}, \theta).$$

- We first rewrite

$$\begin{aligned}\mathcal{L}(Q, \theta) &= \sum_{i=1}^m \sum_{y=1}^k Q_{i,y} \log \left( \frac{P_{\theta}[X = \mathbf{x}_i, Y = y]}{Q_{i,y}} \right) \\&= \sum_{i=1}^m \sum_{y=1}^k Q_{i,y} \log \left( \frac{P_{\theta}[X = \mathbf{x}_i] P_{\theta}[Y = y | X = \mathbf{x}_i]}{Q_{i,y}} \right) \\&= \sum_{i=1}^m \sum_{y=1}^k Q_{i,y} \log P_{\theta}[X = \mathbf{x}_i] + \sum_{i=1}^m \sum_{y=1}^k Q_{i,y} \log \left( \frac{P_{\theta}[Y = y | X = \mathbf{x}_i]}{Q_{i,y}} \right) \\&= L(\theta) - \sum_{i=1}^m KL(Q_y || P_{\theta}[Y = y | X = \mathbf{x}_i]).\end{aligned}$$

- The KL-divergence is non-negative, hence  $\mathcal{L}(Q, \theta)$  is a lower bound for  $L(\theta)$  regardless of the value of  $Q$  used.
- For the particular value of  $Q_y = P_{\theta}[Y = y|X = \mathbf{x}_i]$  the KL-divergence is zero, hence the lower bound is maximized.
  - Setting  $Q_y = P_{\theta}[Y = y|X = \mathbf{x}_i]$  is the E-step in the EM algorithm.
- In fact for this value of  $Q$ ,  $\mathcal{L}(Q^{(t+1)}, \theta^{(t)}) = L(\theta^{(t)})$ . □

# Illustration of EM Algorithm



**Figure:** From [1]. For simplicity, consider a single instance so we do not need to sum over  $m$  instances. We have

$$L(\theta) = \mathcal{L}(Q, \theta) + KL(Q_y || P_{\theta}[Y = y | X = \mathbf{x}]).$$



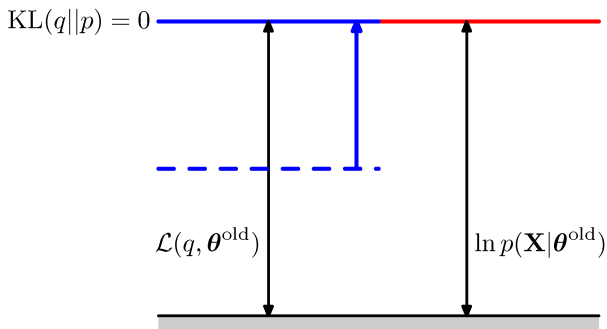
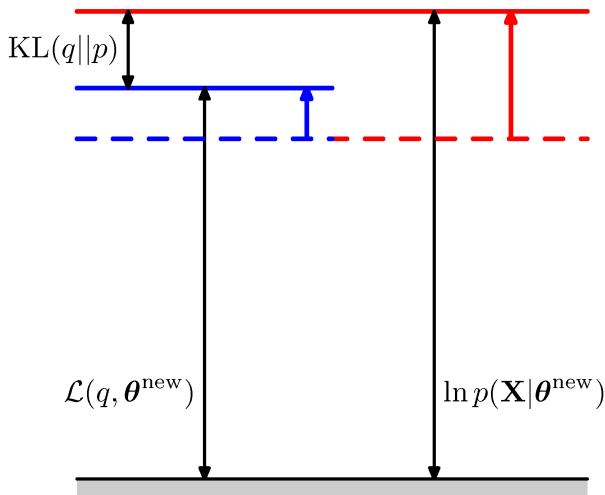


Figure: From [1]. After the E step,  $KL(Q_y||P_{\theta}[Y = y|X = \mathbf{x}]) = 0$ ,  $L(\theta) = \mathcal{L}(Q, \theta)$ .



**Figure:** From [1]. After the M step, again

$KL(Q_y || P_{\theta}[Y = y | X = \mathbf{x}]) \geq 0$ , and we again have

$L(\theta) = \mathcal{L}(Q, \theta) + KL(Q_y || P_{\theta}[Y = y | X = \mathbf{x}])$ , but  $L(\theta)$  is at a higher (or equal) value since  $\mathcal{L}(Q, \theta)$  was maximized in the M step.