Machine Learning

10

10 Machine Learning

10.1 Learning agents

10.2 Inductive learning

10.3 Deep learning

10.4 Reinforcement learning

10.5 Statistical learning$^{+\#}$

10.6 Transfer learning*

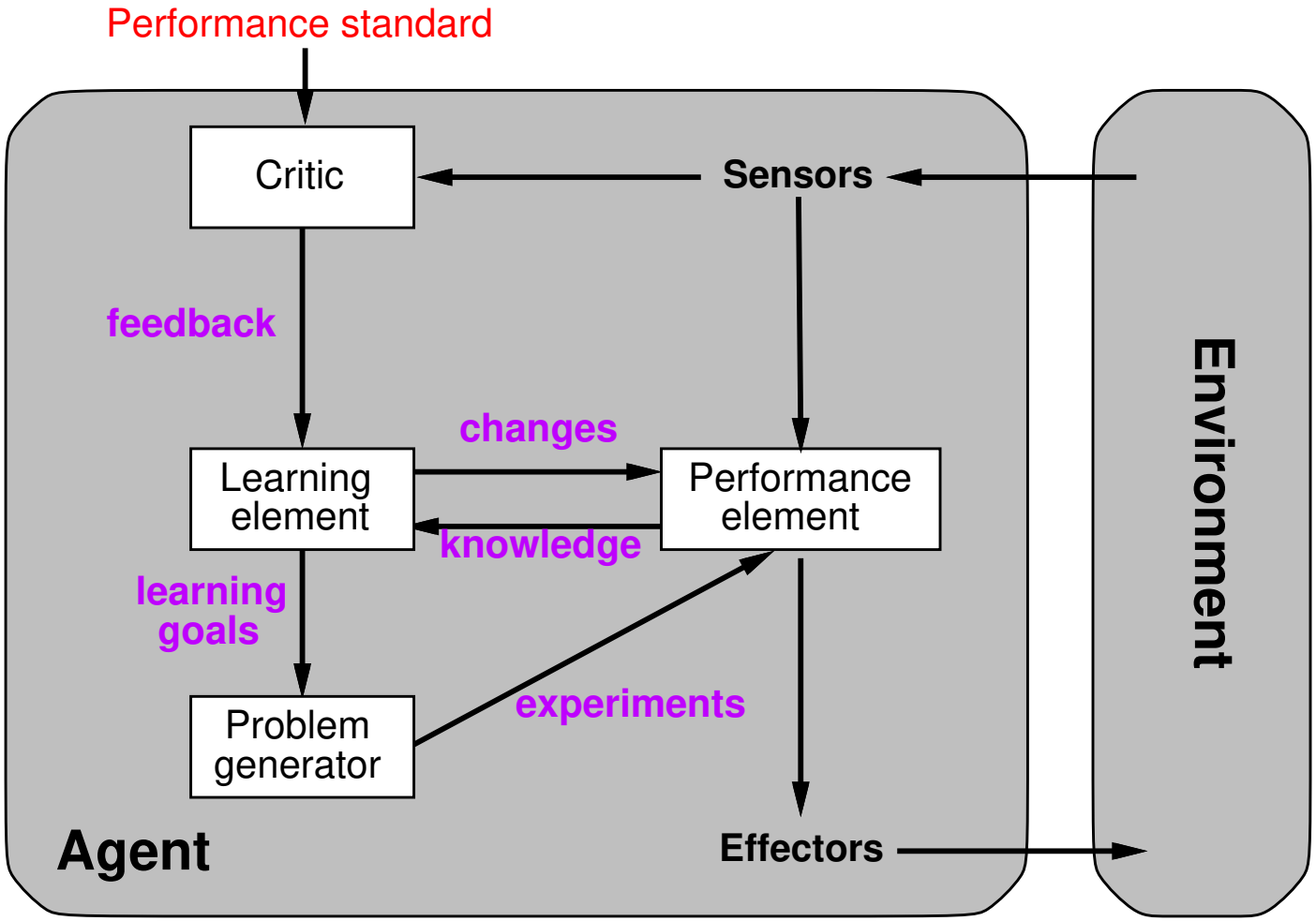
10.7 Ensemble learning*

10.8 Federated learning*

10.9 Explanation-based learning*

10.10 Computational learning theory*

Learning agents



Machine learning

Learning is one basic feature of intelligence
    looking for the principle of learning

Learning is essential for unknown environments
    when a designer lacks omniscience

Learning is useful as a system construction method
    exposing the agent to reality rather than trying to write it down

Learning modifies the agent's decision mechanisms
    improving performance

A.k.a., Data Mining, Knowledge Acquisition (Discovery), Pattern Recognition, Adaptive System, Data Science (Big data) etc.

Types of learning

Supervised learning: correct answers for each example

– requires "teacher" (label)

Unsupervised learning: requires no teacher (unlabeled), but harder

– looking for interesting patterns in examples

– Self-supervised learning: unsupervised learning by unlabelled data in a supervised manner, predicting only a subset of information using the rest

Semisupervised learning: between supervised & unsupervised learning

– improve performance in one of these two by utilizing information associated with the other

Types of learning

Reinforcement learning : occasional rewards

– tries to maximize the rewards

Transfer learning: learning a new task through the transfer of targets from a related source task that has already been learned

Ensemble learning: multiple learners are trained to solve the same problem

Federated learning: many clients collaboratively train a model

Explanation-based Learning: learning in knowledge

Induction

Recall: Induction: if α, β, then $\alpha \to \beta$ (generalization)
(Deduction: if $\alpha, \alpha \to \beta$, then β
Abduction: if $\beta, \alpha \to \beta$, then α)

Induction can be viewed as reasoning or learning

History hint: R. Carnap, The Logical Foundations of Probability, 1950
(induction as probability logic)

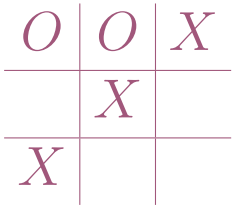
Simplest form: learning (hypotheses) from examples

Math form: learning a function from examples
examples = data, data-driven or adaptive
function = hypothesis/model/parameter, most of applied math
$\Leftarrow$ from philosophy to AI

Inductive learning

f is the target function (task)

An example is a pair x, $f(x)$, e.g., tic-tac-toe



, $+1$

Learning problem: find a hypothesis h

such that $h \approx f$

given examples

$\Rightarrow$ to learn f

Simplified model of human learning

– Ignores explicit knowledge (except for learning in knowledge)

– Assumes examples are **given**

– Assumes that the agent **wants** to learn f

Learning method

To learn f

Find h (output) s.t. $h \approx f$ (approximation/optimizaiton)

given data (input) as a training set

perform well on test set of new data beyond the training set, measuring the accuracy of h

– generalization: the ability to perform well on previously unobserved data

– errors rate (loss/cost): the proportion of mistakes it makes (performance measure)

training error, generalization error, test error

Learning can be simplified as function (curve) fitting

Find a function f^* s.t. $f^* \approx f$

fitting by training data and measuring by test data

Learner

Learner L: a general learning algorithm

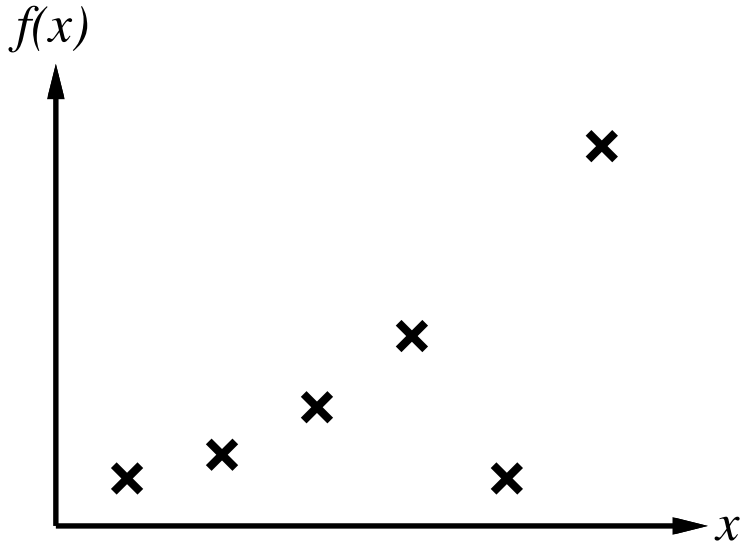
- Task: learning f
- Input: training set $\mathbf{X}$
- Output: approximate function f^*
- Performance measurement (error rate): $\exists \epsilon.\, e < \epsilon,\ f^*(\mathbf{X}) \approx f$

Function fitting

Fit h to agree with f on training set

– h is consistent if it agrees with f on all data

E.g., curve fitting



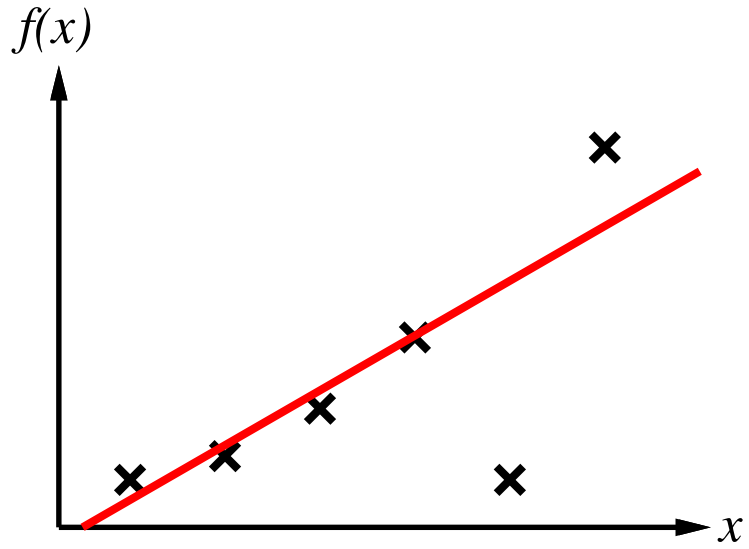
(Inductive) bias: the tendency of a predictive hypothesis to deviate from the expected value when averaged over different training sets

Function fitting

Fit h to agree with f on training set

– underfitting: not able to obtain a low error on the training set

E.g., curve underfitting

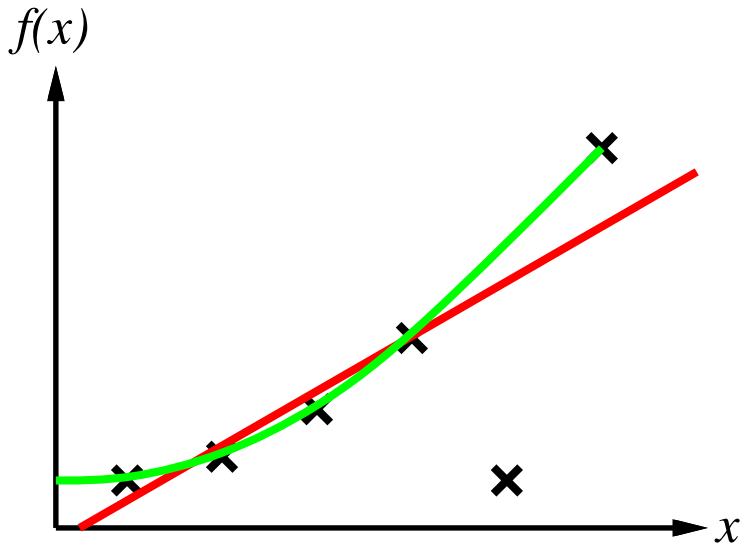


Bias: e.g., the hypothesis space of linear functions induces a strong bias – only allows functions consisting of straight lines

Function fitting

Fit h to agree with f on training set

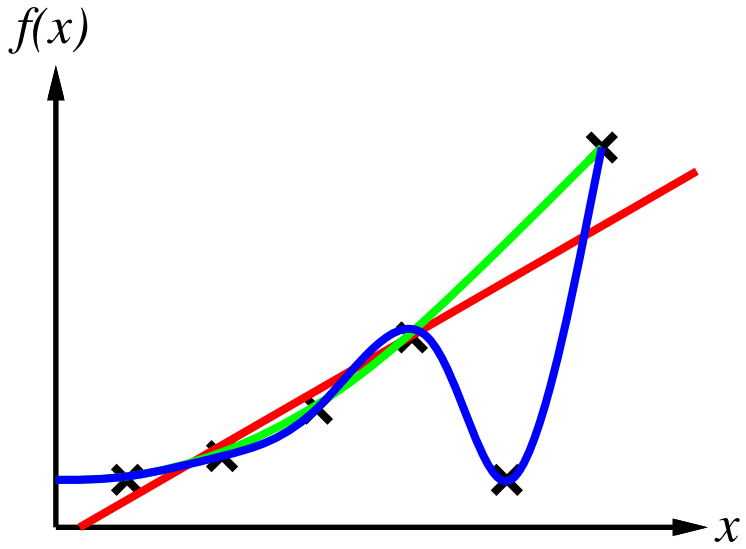
E.g., curve fitting



Variance: the amount of change in the hypothesis due to fluctuation in the training data

Function fitting

Fit h to agree with f on training set
E.g., curve fitting



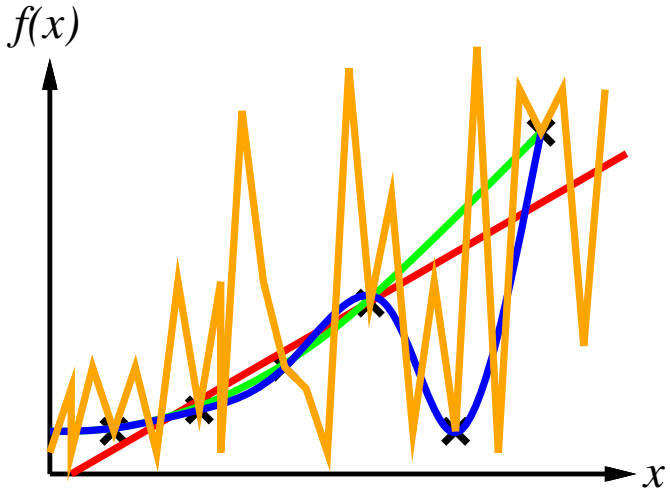
Bias–variance tradeoff between low-bias (complex) hypotheses that fit the training data well and low-variance (simpler) hypotheses that may generalize better

Function fitting

Fit h to agree with f on training set

– overfitting: gap between training error and test error is too large

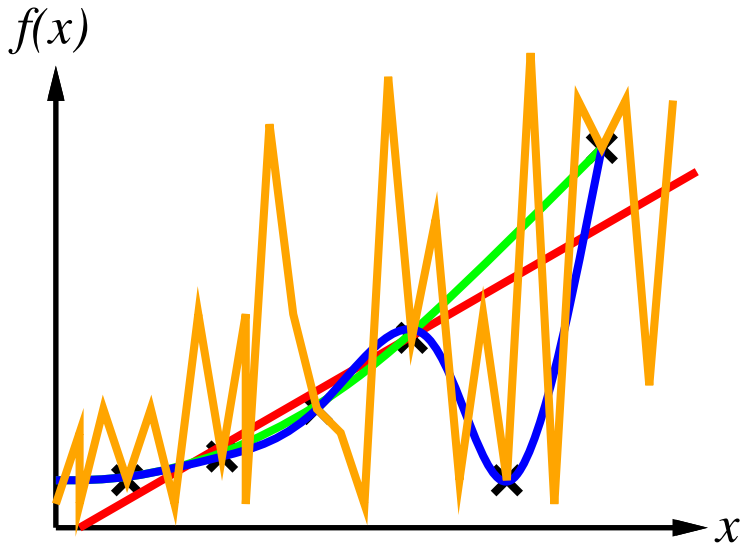
E.g., curve overfitting



Function fitting

Fit h to agree with f on training set

E.g., curve fitting



Ockham's razor: maximize a combination of consistency and simplicity (hard to formalize)

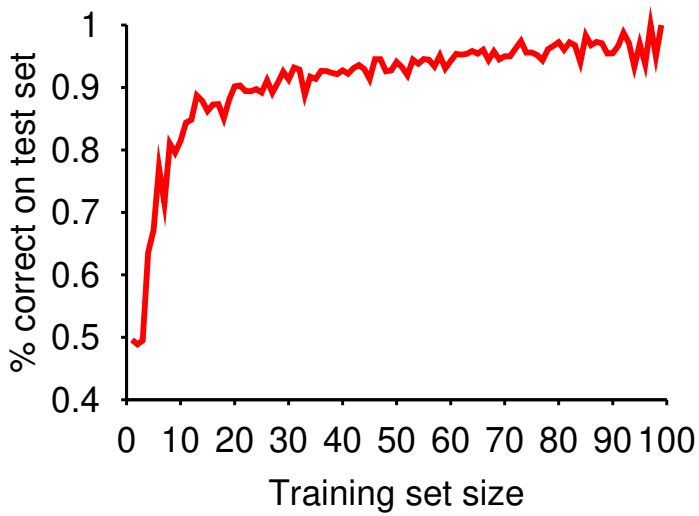
Performance measurement

How do we know that $h \approx f$? (Hume's problem of induction)

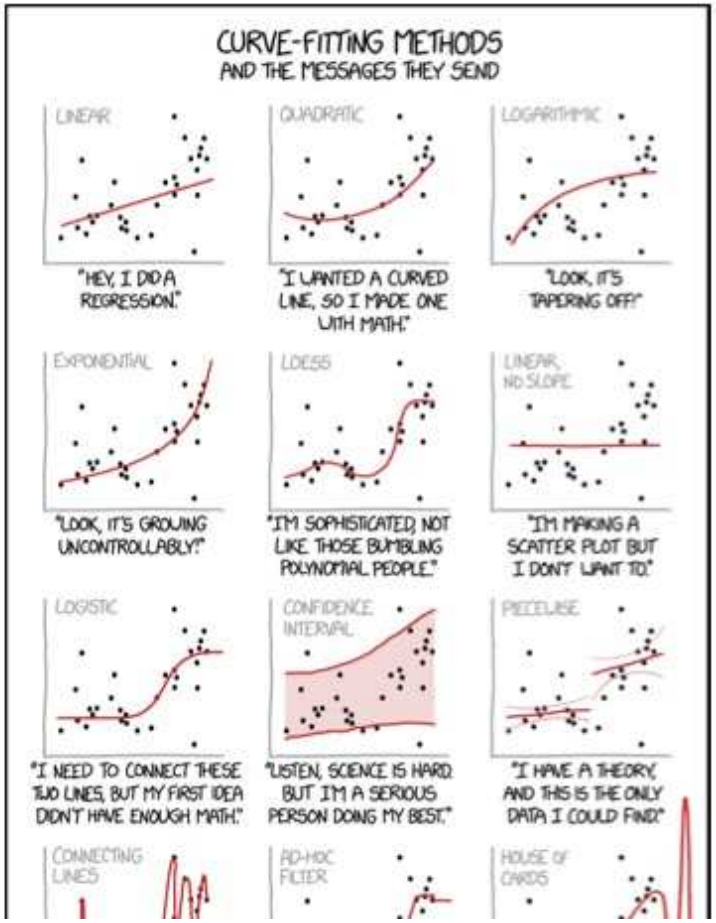
1) Theory, say computational learning theory

2) Try h on a new test set of examples
(use same distribution over example space as training set)

Learning curve = % correct on the test set as a function of training set size



Function fitting



A timely XKCD.com

Attribute-based representations

Examples described by attribute values/features (Boolean, etc.)
E.g., situations where I will/won't wait for a table

Example	Attributes										Target
	Alt	Bar	Fri	Hun	Pat	$Price$	$Rain$	Res	$Type$	Est	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
X_5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*F*	*T*	*French*	*>60*	*F*
X_6	*F*	*T*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Italian*	*0–10*	*T*
X_7	*F*	*T*	*F*	*F*	*None*	*$*	*T*	*F*	*Burger*	*0–10*	*F*
X_8	*F*	*F*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Thai*	*0–10*	*T*
X_9	*F*	*T*	*T*	*F*	*Full*	*$*	*T*	*F*	*Burger*	*>60*	*F*
X_{10}	*T*	*T*	*T*	*T*	*Full*	*$$$*	*F*	*T*	*Italian*	*10–30*	*F*
X_{11}	*F*	*F*	*F*	*F*	*None*	*$*	*F*	*F*	*Thai*	*0–10*	*F*
X_{12}	*T*	*T*	*T*	*T*	*Full*	*$*	*F*	*F*	*Burger*	*30–60*	*T*

Classification of examples is positive (T) or negative (F)

Decision trees learning

DT (Decision Trees): supervised learning

Training set: input data (examples) and corresponding labels(/ground-truth/targets/answer) t

- Regression: t is a real number (e.g., stock price)
- Classification: t is an element of a discrete set $\{1, \ldots, C\}$

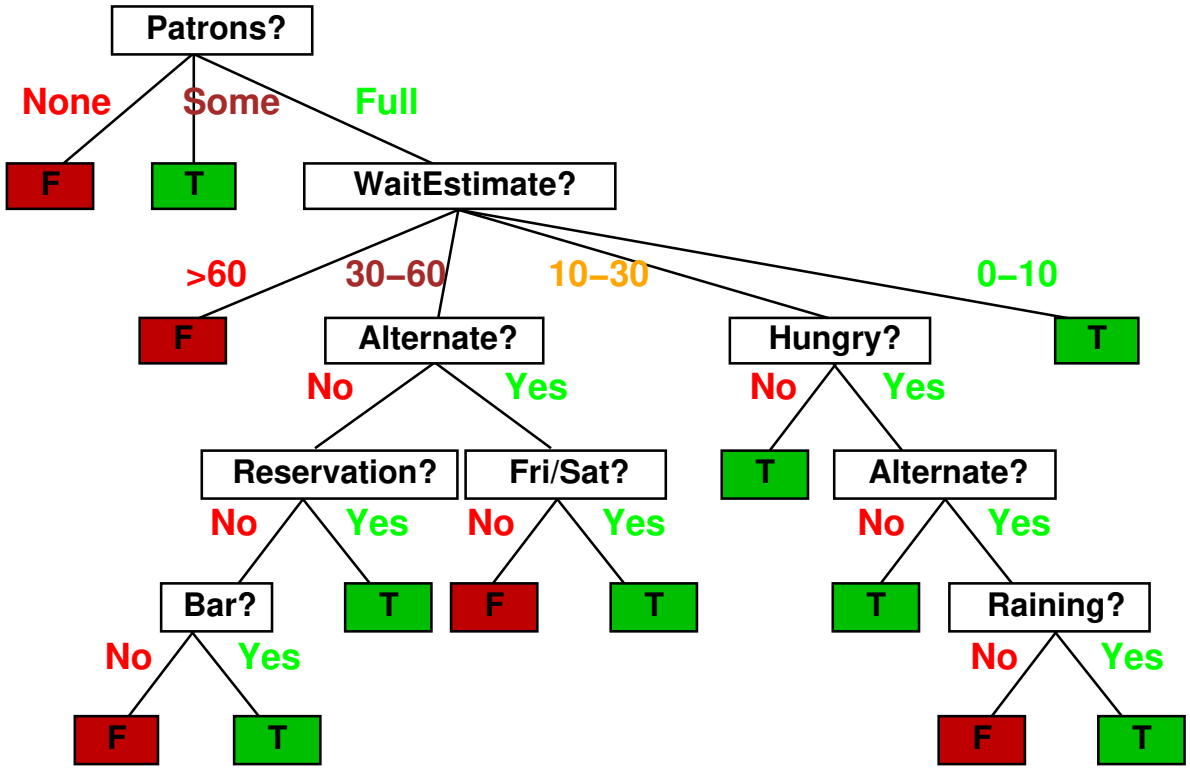
  t is often a highly structured object (e.g., image)

  Binary classification: t is T or F $\Leftarrow$ simple DT

f takes input features and returns output ("decision") as trees

Decision trees

E.g., here is the "true" tree for deciding whether to wait for
one possible representation for hypotheses to be induced

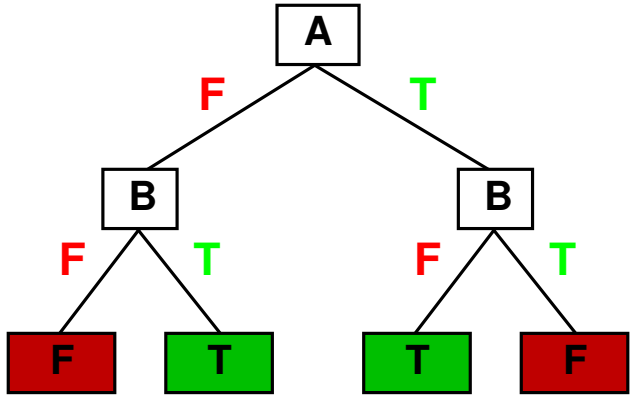


Expressiveness

DT induction is the simplest and yet most successful form of learning
can express any function of the input attributes

E.g., for Boolean functions, truth table row $\rightarrow$ path to leaf

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F



Prefer to find more **compact** decision trees

Expressiveness*

- Discrete-input, discrete-output case
  Decision trees can express any function of the input attributes

- Continuous-input, continuous-output case
  Decision trees can approximate any function arbitrarily closely

Trivially, there is a consistent decision tree for any training set
w/ one path to leaf for each example
(unless f nondeterministic in $\mathbf{x}$)
but it probably won't generalize to new examples

Hypothesis spaces

How many distinct decision trees with n Boolean attributes??

Hypothesis spaces

How many distinct decision trees with n Boolean attributes??

= number of Boolean functions

Hypothesis spaces

How many distinct decision trees with n Boolean attributes??

= number of Boolean functions
= number of distinct truth tables with 2^n rows

Hypothesis spaces

How many distinct decision trees with n Boolean attributes??

= number of Boolean functions

= number of distinct truth tables with 2^n rows = 2^{2^n}

Hypothesis spaces

How many distinct decision trees with n Boolean attributes??

= number of Boolean functions
= number of distinct truth tables with 2^n rows = 2^{2^n}

E.g., with 6 Boolean attributes, there are 18,446,744,073,709,551,616 trees

Hypothesis spaces

How many distinct decision trees with n Boolean attributes??

$=$ number of Boolean functions
$=$ number of distinct truth tables with 2^n rows $= 2^{2^n}$

E.g., with 6 Boolean attributes, there are 18,446,744,073,709,551,616 trees

How many purely conjunctive hypotheses (e.g., $Hungry \wedge \neg Rain$)??

Hypothesis spaces

How many distinct decision trees with n Boolean attributes??

= number of Boolean functions
= number of distinct truth tables with 2^n rows = 2^{2^n}

E.g., with 6 Boolean attributes, there are 18,446,744,073,709,551,616 trees

How many purely conjunctive hypotheses (e.g., $Hungry \wedge \neg Rain$)??

Each attribute can be in (positive), in (negative), or out
$\Rightarrow 3^n$ distinct conjunctive hypotheses

More expressive hypothesis space
- increases the chance that target function can be expressed
- increases the number of hypotheses consistent w/ training set
  $\Rightarrow$ may get worse predictions

DT learning

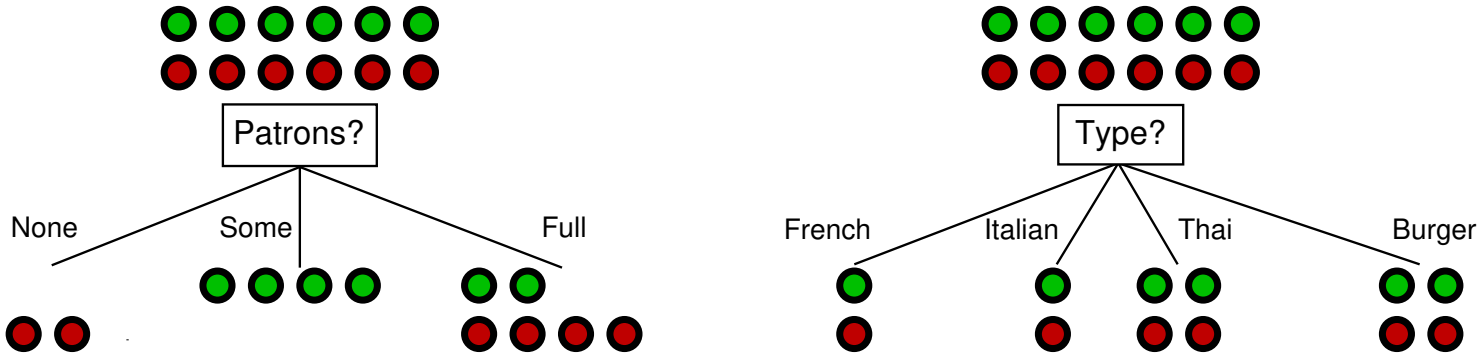
Aim: find a small tree consistent with the training examples

Idea: (recursively) choose "most significant" attribute as root of (sub)tree

```
def DTL(examples, attributes, parent-examples)
    if examples is empty then return PLURALITY-VALUE(parent-examples)
    else if all examples have the same classification then return the classification
    else if attributes is empty then return PLURALITY-VALUE(parent-examples)
    else
        A ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
        tree ← a new decision tree with root test A
        for each value v of A do
            exs ← {e : e ∈ examples and e.A = v}
            subtree ← DTL(exs, attributes - A, examples)
            add a branch to tree with label (A = v) and subtree subtree
        return tree
```

Choosing an attribute

Idea: a good attribute (IMPORTANCE) splits the examples into subsets that are (ideally) "all positive" or "all negative"



Patrons? is a better choice—gives **information** about the classification

Information#

Information answers questions

The more clueless I am about the answer initially, the more information is contained in the answer

Scale: 1 bit = answer to Boolean question with prior $\langle 0.5, 0.5 \rangle$

Information in an answer when prior is $\langle P_1, \ldots, P_n \rangle$ is

$$H(\langle P_1, \ldots, P_n \rangle) = \sum_{i=1}^{n} - P_i \log_2 P_i$$

(called entropy of the prior)

Information#

Suppose we have p positive and n negative examples at the root

$\Rightarrow H(\langle p/(p+n), n/(p+n)\rangle)$ bits needed to classify a new example

E.g., for 12 restaurant examples, $p = n = 6$ so we need 1 bit

An attribute splits the examples E into subsets E_i, each of which (we hope) needs less information to complete the classification

Information#

Let E_i have p_i positive and n_i negative examples

$\Rightarrow H(\langle p_i/(p_i+n_i), n_i/(p_i+n_i)\rangle)$ bits needed to classify a new example

$\Rightarrow$ **expected** number of bits per example over all branches is

$$\sum_i \frac{p_i+n_i}{p+n} H(\langle p_i/(p_i+n_i), n_i/(p_i+n_i)\rangle)$$

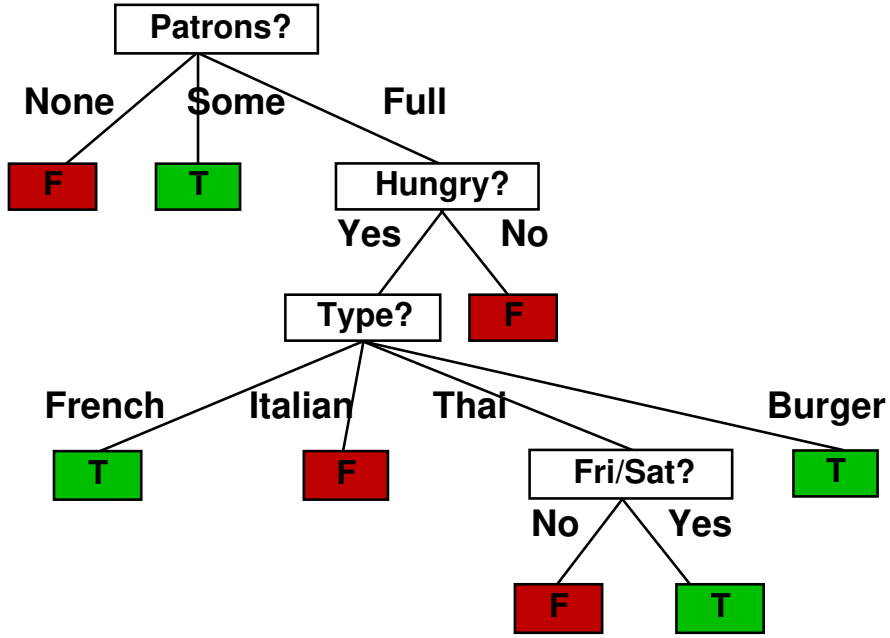
For $Patrons?$, this is 0.459 bits, for $Type$ this is (still) 1 bit

choose the attribute that minimizes the remaining information needed

$\Rightarrow$ just what we need to implement IMPORTANCE

Example: decision tree

Decision tree learned from the 12 examples



Substantially simpler than the original tree — with more training examples some mistakes could be corrected

DT: classification and regression*

DT can be extended

each path from the root to a leaf defines a region of input space

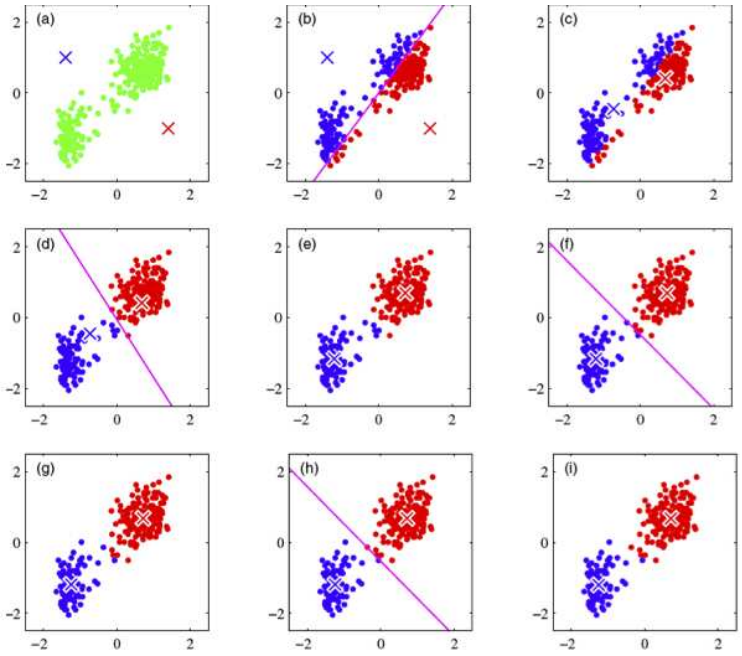
- Classification tree: discrete output
  leaf value is typically set to the most common value in a class set
- Regression tree: continuous output
  leaf value is typically set to the mean value in a class set

K-means learning

K-means: unsupervised learning

have some unlabbled data, and want to infer the causal structure underlying the data — the structure is latent, i.e., never observed

Clustering: grouping data points into clusters



K-means

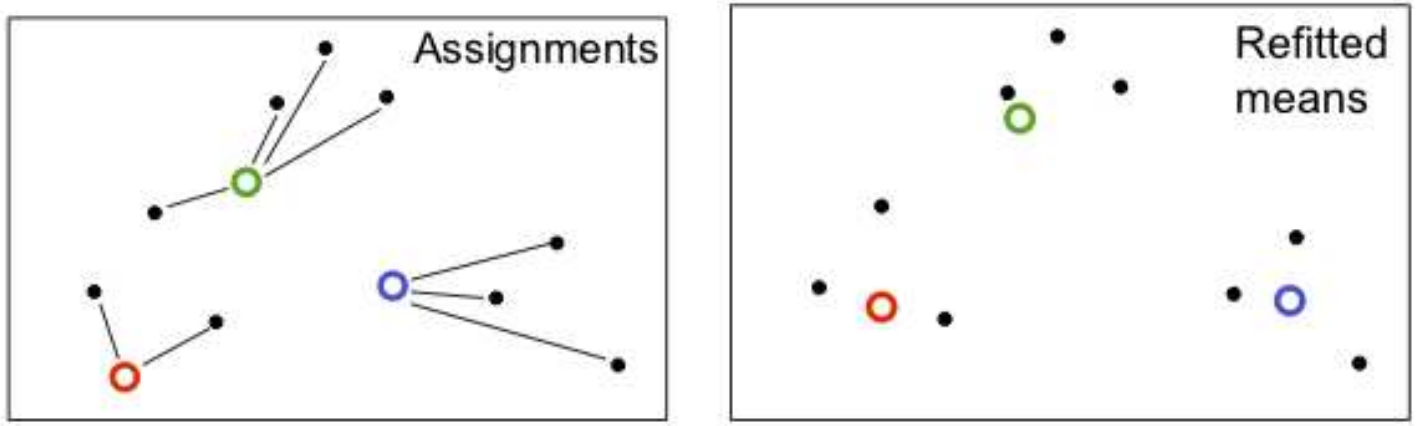
Idea

- Assumes there are k clusters, and each point is close to its cluster center (the mean of points in the cluster)
- If we knew the cluster assignment we could easily compute means
- If we knew the means we could easily compute cluster assignment
- Chicken and egg problem
- Can show it is NP hard
- Very simple (and useful) heuristic — start randomly and alternate between the two

K-means learning#

1. **Initialization**: randomly initialize cluster centers

2. Iteratively alternates between two steps
   - **Assignment step**: Assign each data point to the closest cluster
   - **Refitting step**: Move each cluster center to the center of gravity of the data assigned to it



K-means algorithm#

1. **Initialization**: Set K cluster means $\mathbf{m}_1, \ldots, m_K$ to random values

2. Repeat until convergence (until assignments do not change)
   - **Assignment**: Each data point $\mathbf{x}^{(n)}$ assigned to nearest mean
     $\hat{h}^n = arg\, min_k\, d(\mathbf{m}_k, \mathbf{x}^{(n)})$
     (with, e.g., L2 norm: $\hat{h}^n = arg\, min_k\, ||\mathbf{m}_k - \mathbf{x}^{(n)}||$)
     and **Responsibilities** (1-hot encoding)
     $\hat{r}_k^{(n)} = 1 \leftrightarrow \hat{k}^{(n)} = k$
   - **Refitting**: Model parameters, means are adjusted to match sample means of data points they are responsible for
     $$\mathbf{m}_k = \frac{\sum_n r_k^{(n)} \mathbf{x}^{(n)}}{\sum_n r_k^{(n)}}$$

Hyperparameter

Hyperparameter: choosing k by fine-tuning (no learning)

Hyperparameters

- are settings to control the algorithm's behavior
- most of the learning has the hyperparameters
- tuning: hand-tuning, grid search, random search (sampling)
- can be learned as well (nested learning procedure)

Validation

Validation set: divide the available data (without the test set) into a training set and a validation set

– lock the test set away until the learning is done to obtain an independent evaluation of the final hypothesis

Can tune hyperparameters using a validation set

Measure the generalization error (error rate on new examples) using a test set

Usually, the dataset is partitioned

training set $\cup$ validation set $\cup$ test set

training set $\cap$ validation set $\cap$ test set $= \{\}$

Self-learning

Self-learning/pseudo-labelling (wrapper method): semi-supervised learning

Idea: a single supervised classifier that is iteratively trained on both labeled data and data that has been pseudo-labeled in previous iterations of the algorithm

Given one (or more) supervised classifier(s) as base leaner, say DT/KNN, self-learning usually consists of two alternating steps

1. **Training**: The base learner is trained on the labeled data and possibly pseudo-labeled data from previous iterations

2. **Pseudo-labelling**: The resulting classifier is used to infer labels for the previously unlabelled data, and the data for which the learner was most confident of their predictions are pseudo-labeled for use in the next iteration

Pseudo-labelling#

Iteratively training with the original labeled data and previously unlabelled data that is augmented with predictions from earlier iterations of the learner, the latter is referred to as pseudo-labeled data

The prediction for pseudo-labels can be done in an optimal way like $argmin$ (loss function, see below)

Hint: there are intrinsically semi-supervised extensions of many supervised learning approaches

Regression

Learner L_R: Regression

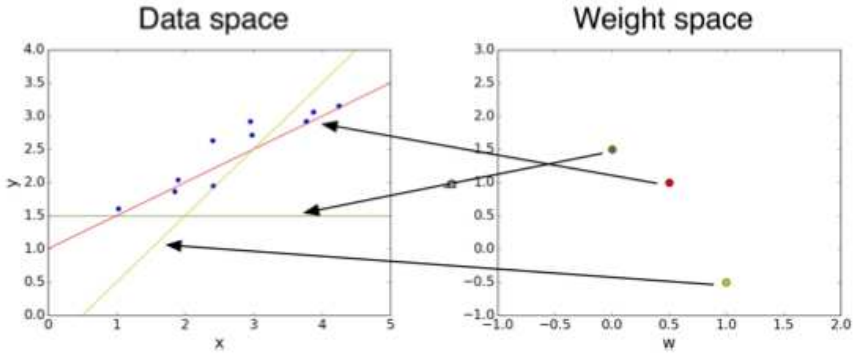
Algorithm

- choose a model describing the relationships between variables of interest
- define a loss function quantifying how bad is the fit to the data
- choose a regularizer saying how much we prefer different candidate explanations
- fit the model, e.g. using an optimization algorithm

-Parametric model: a learning model that summarizes data with a set of parameters of fixed size
-Nonparametric model: a learning model that cannot be characterized by a bounded set of parameters

Regression problem

Want to predict a scalar t as a function of a scalar x
Given a dataset of pairs (inputs, targets/labels) $\{(\mathbf{x}^{(i)}, t^{(i)})\}_{i=1}^{N}$



Linear regression model (linear model): an affine (linear) function
$y = wx + b$

- y is the prediction
- w is the weight
- b is the bias
- w and b together are the parameters (parametric model)
- Settings of the parameters are called hypotheses

Loss function

Loss function: squared error SE (says how bad the fit is)

$$\mathcal{L}(y,t)=\frac{1}{2}(y-t)^2$$

$y-t$ is the residual, and want to make this small in magnitude
(the $\frac{1}{2}$ factor is just to make the calculations convenient)

Cost function: loss function averaged over all training examples

$$\mathcal{J}(w,b)=\frac{1}{2N}\sum_{i=1}^{N}(y^{(i)}-t^{(i)})^2=\frac{1}{2N}\sum_{i=1}^{N}(wx^{(i)}+b-t^{(i)})^2$$

Multivariable regression: linear function

$$y=\sum_{j=1}^{d}w_jx_j+b=\mathbf{w}^\top\mathbf{x}+b$$

$\mathbf{x}\in\mathbb{R}^d$, d is the dimension of $\mathbf{x}$

(no different than the single input case, just harder to visualize)

Optimization problem

Optimization: minimize cost function, i.e. finding the parameters $\mathbf{w}^*, b^*$ s.t.

$$\mathbf{w}^*, b^* = \underset{\mathbf{w},b}{\operatorname{argmin}} \mathcal{J}(\mathbf{w}, b)$$

- Direct solution: minimum of a smooth function (if it exists) occurs at a critical point, i.e., the point where the derivative is zero

  Linear regression is one of only a handful of models that permit direct solution

- Gradient descent (GD): an iteration (algorithm) by applying an update repeatedly until some criterion is met

  Initialize the weights to something reasonable (e.g., all zeros) and repeatedly adjust them in the direction of steepest descent

Closed form solution#

- Chain rule for derivatives

$$\frac{\partial \mathcal{L}}{\partial w_j} = (y - t)x_j \qquad \frac{\partial \mathcal{L}}{\partial b} = y - t$$

- Cost derivatives

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - t^{(i)})x_j^{(i)}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{N}\sum_{i=1}^{N} y^{(i)} - t^{(i)}$$

- Analytic in matrix $\mathbf{y} \in \mathbb{R}^n$

$$\mathbf{w}^* = \left(\mathbf{x}^\top \mathbf{x}\right)^{-1} \mathbf{x}^\top \mathbf{y}$$

Gradient descent

Known

if $\frac{\partial \mathcal{J}}{\partial w_j} > 0$, then increasing w_j increases $\mathcal{J}$

if $\frac{\partial \mathcal{J}}{\partial w_j} < 0$, then increasing w_j decreases $\mathcal{J}$

Updating: decreases the cost function

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j} = \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) x_j^{(i)}$$

$$b \leftarrow b - \alpha \frac{\partial \mathcal{J}}{\partial w_j} = \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)})$$

α is a learning rate (hyperparameter): the larger it is, the faster $\mathbf{w}$ changes

typically small, e.g., 0.01 or 0.0001

Stochastic gradient descent

GD: the loss surface is convex

– there is no local minima

– convergence to the global minimum

But, very slow

• Batch GD: sum over all N training examples for every epoch (each step that covers all the training examples)

• Stochastic gradient descent (SGD): randomly selects a minibatch of m out of the N examples at each step $\Leftarrow$ faster

– say, $m = 100$, $N = 10,000$, reduced by a factor of 100

– since the error of the gradient is proportional to the square root of the number of examples, the error increases by only a factor of 10

– take 10 times more steps before convergence, minibatch SGD is 10 times faster than a full batch

Gradient descent vs closed form solution#

- GD can be applied to a much broader set of models
- GD can be easier to implement than direct solutions, especially with automatic differentiation software
- For regression in high-dimensional spaces, GD is more efficient than direct solution (matrix inversion is an $\mathcal{O}(d^3)$ algorithm)

Hints

- For-loops in Python are slow, so we vectorize algorithms by expressing them in terms of vectors and matrices
- Vectorized code is much faster
- Matrix multiplication is very fast on a GPU (Graphics Processing Unit)

Maximum likelihood estimators#

Usually, optimizing the cost by maximizing the likelihood (MLE)

$p(\mathbf{y} \mid \mathbf{x})$

Consider linear regression by assuming that observations arise from noisy ϵ which is a normal (Gauss) distribution

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon, \epsilon \sim \mathcal{N}\left(0, \sigma^2\right)$$

Recall Gauss (normal) distribution with mean μ and variance σ^2 (standard deviation σ)

$$\mathcal{N}(\mu, \sigma) = p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right)$$

Maximum likelihood estimators#

Write out the likelihood of seeing y for given $\mathbf{x}$

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}\left(y - \mathbf{w}^\top \mathbf{x} - b\right)^2\right)$$

While maximizing the product of exponential functions might look difficult, maximizing the log of the likelihood instead; more often expressed as minimization, i.e., minimize the negative log-likelihood

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2}\log\left(2\pi\sigma^2\right) + \frac{1}{2\sigma^2}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2$$

Just need one more assumption that σ is some fixed constant, the first term can be ignored and the second term is identical to the squared error loss except for the multiplicative constant

Theorem minimizing the mean SE = maximizing MLE
(of a linear model under the additive Gaussian noise)

Cross entropy loss#

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P\left(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}\right) = \sum_{i=1}^{n} l\left(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}\right)$$

where for any pair of label $\mathbf{y}^{(i)}$ and the prediction $\hat{\mathbf{y}^{(i)}}$ over q classes, the loss function l is called the cross-entropy

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j$$

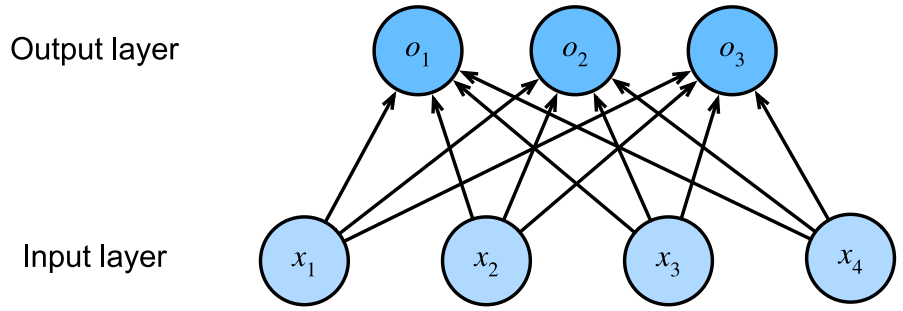
Softmax regression#

Regression ⇒ Classification

Want to predict 4 features and 3 output categories

$$o_1 = x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1$$
$$o_2 = x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2$$
$$o_3 = x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3$$
$$\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Regression is a fully connected network, the same as a single-layer neural network (see later)



Softmax regression#

Softmax function: transforming the logits o_j nonnegative and sum to 1, while requiring the model differentiable

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$$

Softmax in the loss

$$\begin{aligned} l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} \\ &= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \\ &= \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \end{aligned}$$

Softmax regression#

Derivative w.r.t. any logit o_j

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j$$

Remark: *log-it*

$$logit = log(Odds) = log(\frac{P}{1-P})$$

$$logit\,(P_i) = \ln \frac{P_i}{1-P_i} = w_1 x_{i1} + w_2 x_{i2} + \cdots + w_n x_{in} + b$$

Predicting prob. (distribution) P for linear regression

Classification

- Classification: predict a discrete-valued target
- Binary: predict a binary target $t \in \{0, 1\}$
- Linear: model is a linear function of $\mathbf{x}$, followed by a threshold

$$z = \mathbf{w}^\top \mathbf{x} + b$$

$$y = \begin{cases} 1 & \text{if } z \geq r \\ 0 & \text{if } z < r \end{cases}$$

Linear classification

Simplification: eliminating the threshold and the bias

- Assume (without loss of generality) that $r = 0$

$$\mathbf{w}^{\mathrm{T}}\mathbf{x} + b \geq r \Longleftrightarrow \mathbf{w}^{\mathrm{T}}\mathbf{x} + \underbrace{b - r}_{\triangleq b'} \geq 0$$

- Add a dummy feature x_0 which always takes the value 1, and the weight w_0 is equivalent to a bias

Simplified model

$$z = \mathbf{w}^{\mathrm{T}}\mathbf{x}$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Examples

NOT

x_0	x_1	t
1	0	1
1	1	0

$$b > 0$$
$$b + w < 0$$
$$b = 1, w = -2$$

Examples

AND

x_0	x_1	x_2	t
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

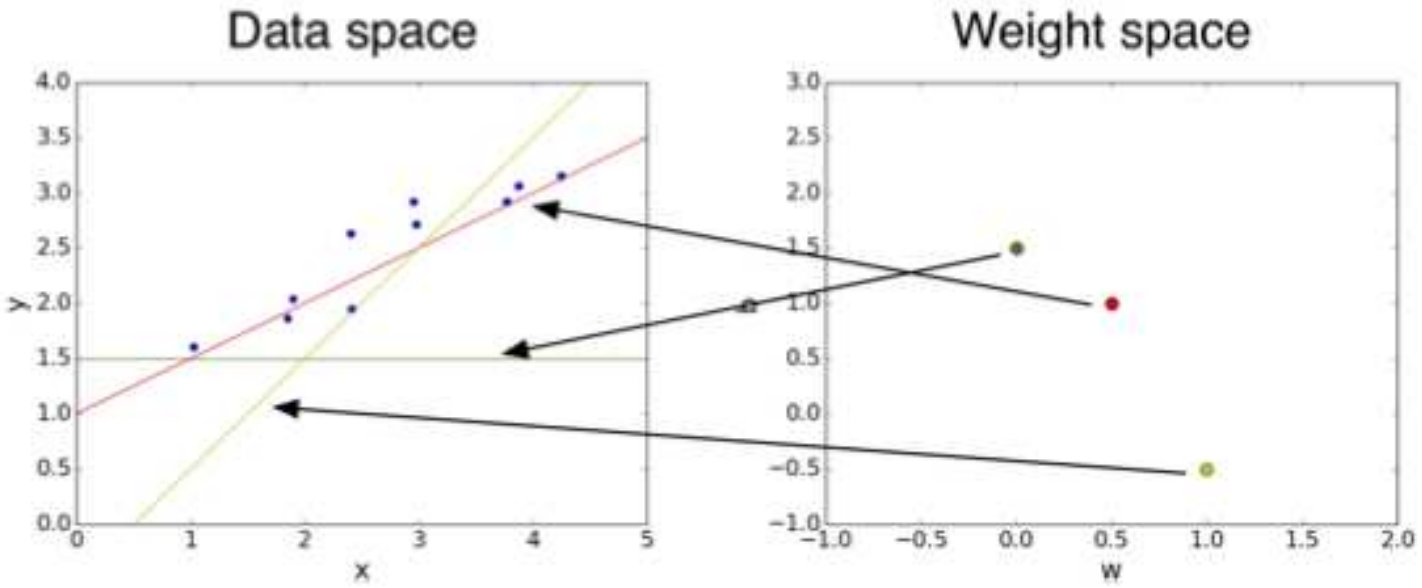
$$
\begin{aligned}
b &< 0 \\
b + w_2 &< 0 \\
b + w_1 &< 0 \\
b + w_1 + w_2 &> 0
\end{aligned}
$$

$$b = -1.5, w_1 = 1, w_2 = 1$$

Question: Can a binary linear classification simulate propositional connectives (propositional logic)?

The geometric interpretation

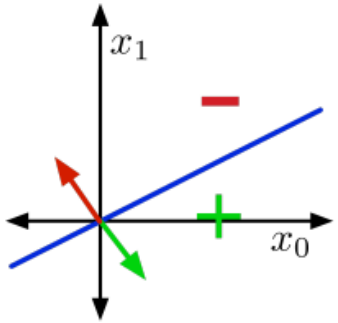
Recall from linear regression



Say, calculating the NOT/AND weight space

The geometric interpretation#

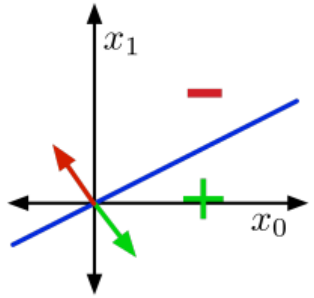
Input Space (data space)



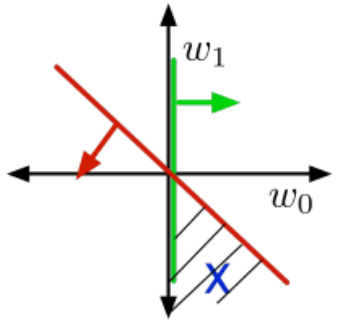
- Visualizing the **NOT** example
- Training examples are points
- Hypotheses are half-spaces whose boundaries pass through the origin (the point $f(x_0, x_1)$ in the half-space)
- The boundary is the decision boundary
  - In 2D, it's a line, but think of it as a hyperplane
- The training examples are linearly separable

The geometric interpretation

Weight Space



$w_0 > 0$

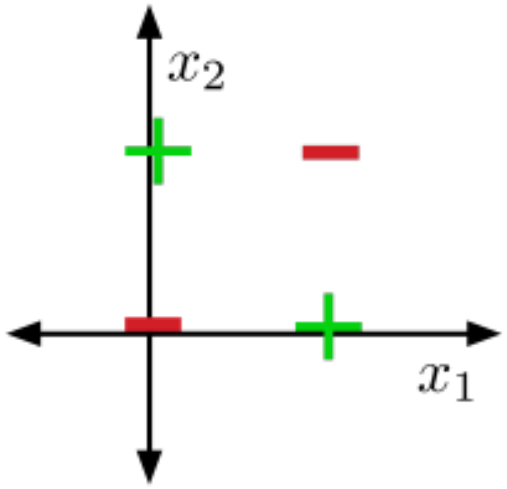


$w_0 + w_1 < 0$

Limits of linear classification

Linearity implies monotonicity: any increase/decrease in the input must either cause an increase/decrease in the output (if the weight is positive/negative)

Some datasets are not linearly separable, e.g., **XOR**



XOR is not linearly separable

Limits of linear classification

• Sometimes we can overcome this limitation using feature maps, just like for linear regression, e.g., **XOR**

$$\phi(\mathbf{x}) = \begin{pmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{pmatrix}$$

x_1	x_2	$\phi_1(\mathbf{x})$	$\phi_2(\mathbf{x})$	$\phi_3(\mathbf{x})$	t
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	0	1
1	1	1	1	1	0

• This is linearly separable $\Leftarrow$ Try it

• Not a general solution: it can be hard to pick good basis functions

• Instead, neural networks can be used as a general solution to learn nonlinear hypotheses directly

Cross validation*

Want to learn the best hypothesis (choosing and evaluating)

– assumption: independent and identically distributed (i.i.d.) example space

i.e., there is a probability distribution over examples that remains stationary over time

Cross-validation (Larson, 1931): randomly split the available data into a training set and a test set

– fails to use all the available data

– invalidates the results by inadvertently **peeking** at the test data

Cross validation

k-fold cross-validation: each example serves as training data and test data

- splitting the data into k equal subsets
- performing k rounds of learning
  – on each round $1/k$ of the data is held out as a test set and the remaining examples are used as training data

The average test set score of the k rounds should be a better estimate than a single score

  – popular values for k are 5 and 10

Cross validation

```
def CROSS-VALIDATION(Learner,size,k,examples)
    local variables: errT, an array, indexed by size, storing training-set error rates
                     errV, an array, indexed by size, storing validation-set error rates

    fold-errT ← 0; fold-errV ← 0
    for fold = 1 to k do
    training set,validation set ← PARTITION(examples,fold,k)
    h ← Learner(size,training set)
    fold-errT ← fold-errT + ERROR-RATE(h, training set)
    fold-errV ← fold-errV +ERROR-RATE(h, validation   set)
    return fold-errT/k,fold-errV/k
    // Two values: average training set error rate, average validation set error rate
```

Model selection*

Complexity versus goodness of fit

select among models that are parameterized by size

for decision trees, the size could be the number of nodes in the tree

Wrapper: takes a learning algorithm as an argument (e.g., DT)

– enumerates models according to a parameter, *size*

– – for each size, uses cross-validation on *Learner* to compute the average error rate on the training and test sets

– starts with the smallest, simplest models (probably underfit the data), and iterates, considering more complex models at each step, until the models start to overfit

Model selection

```
def CROSS-VALIDATION-WRAPPER(Learner,k,examples)
    local variables: errT, errV

    for size = 1 to ∞ do
        errT [size], errV [size] ← CROSS-VALIDATION(Learne r, size, k, examples)
        if errT has converged then do
            best size ← the value of size with minimum errV [size]
    return Learner(best size, examples)
```

Simpler form of meta-learning: learning what to learn

Generalization*

Recall: generalization is the ability to perform well on previously unobserved data

Problem: if consistency with the training examples is considered an appropriate generalization, then a learner can never make the induction beyond those it has observed

Only if the learner has other sources of information (or biases) for choosing one generalization over the other, can it non-arbitrarily classify instances beyond those in the training set

Inductive bias

Inductive bias: allows a learning algorithm to prioritize one solution (or interpretation) over another, independent of the observed data

- assumptions about either the data-generating process
- the prior distribution (Bayesian model)
- a regularization term (added to avoid overfitting)
- the architecture of the algorithm (neural network), etc.

If unbiased generalization systems are incapable of making the inductive leap to characterize the new instances, then the power of a generalization system follows directly from its biases – from decisions based on criteria other than consistency with the training instances
Ideally, inductive biases both improve the search for solutions without substantially diminishing performance and help find solutions that generalize in a desirable way
Understanding learning mechanisms depends upon understanding various biases

Regularization*

From error rates to loss function: $l(x, y, \widehat{y}) \approx l(y, \widehat{y})$

$=$ Utility (result of using y given an input x)

- Utility (result of using $\widehat{y}$ given an input x)

amount of utility lost by predicting $h(x) = \widehat{y}$ when the correct answer is $f(x) = y$

e.g., it is worse to classify non-spam as spam than to classify spam as non-spam

Regularization (for a function that is more regular, or less complex): an alternative approach to search for a hypothesis

directly minimizes the weighted sum of the loss and the complexity of the hypothesis (total cost)

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error

Deep learning

Artificial Neural Networks (ANNs or NNs), also known as

- connectionism
- parallel distributed processing (PDP)
- neural computation
- computational neuroscience
- representation learning
- **deep learning**

have basic ability to learn

Applications: pattern **recognition** (speech, handwriting, object) , driving and fraud detection etc.

A brief history of neural networks#

300B.C.	Aristotle	Associationism, attempt. to understand brain
1873	Bain	Neural Groupings (inspired Hebbian Rule)
1936	Rashevsky	Math model of neutrons
1943	McCulloch/Pitts	MCP Model (ancestor of ANN)
1949	Hebb	founder of NNs, Hebbian Learning Rule
1958	Rosenblatt	Perceptron
1974	Werbos	Backpropagation
1980	Kohonen	Self Organizing Map
	Fukushima	Neocogitron (inspired CNN)
1982	Hopfield	Hopfield Network
1985	Hilton/Sejnowski	Boltzmann Machine
1986	Smolensky	Harmonium (Restricted Boltzmann Machine)
	Jordan	Recurrent Neural Network
1990	LeCun	LeNet (deep networks in practice)
1997	Schuster/Paliwal	Bidirectional Recurrent Neural Network
	Hochreiter/Schmidhuber	LSTM (solved vanishing gradient)

A brief history of neural networks#

2006	Hilton	Deep Belief Networks, opened deep learning era
2009	Salakhutdinov/Hinton	Deep Boltzmann Machines
2012	Hinton	Dropout (efficient training), AlexNet

History reminder

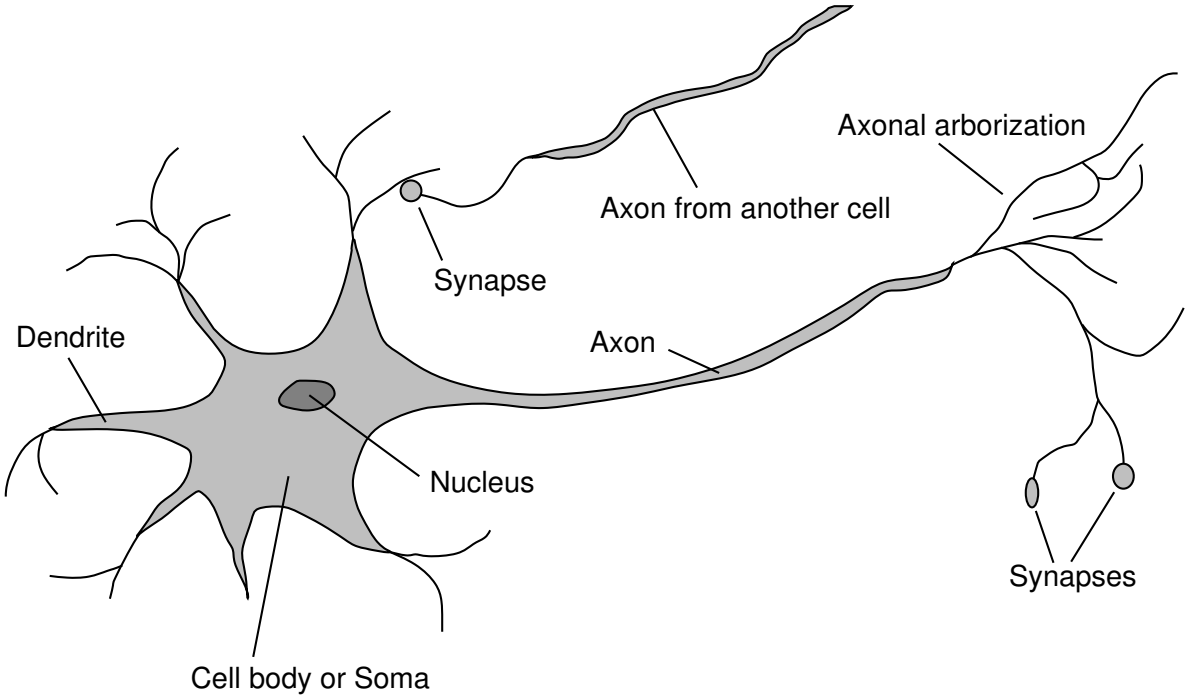
- known as ANN (and cybernetics) in the 1940s – 1960s
- connectionism in the 1980s – 1990s
- resurgence under the name deep learning beginning in 2006

Moving conditons:

– Increasing dataset sizes
– Increasing network sizes (computational resources)
– Increasing accuracy, complexity and impact in applications

Brains#

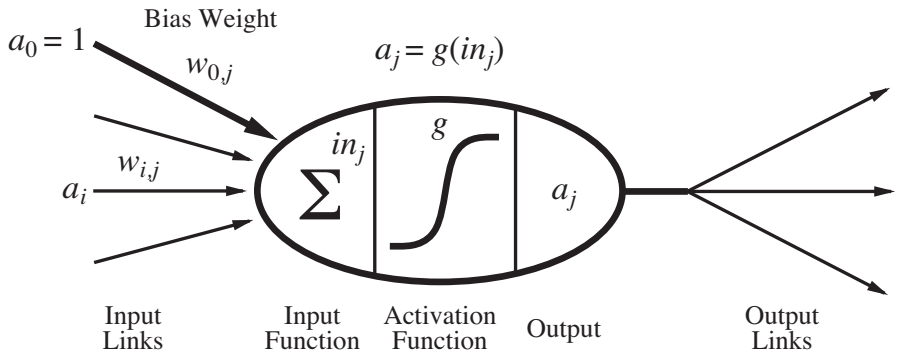
Recall: 10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time. Signals are noisy "spike trains" of electrical potential



McCulloch–Pitts "neuron"

Output is a nonlinear function (activation) of the inputs

$$a_j \equiv g(in_j) = g\left(\sum_i W_{i,j} a_i\right) = g\left(\mathbf{W}^\top \mathbf{a}\right)$$



A neural network (NN) is a collection of units (neurons) connected by **directed** links (graph)

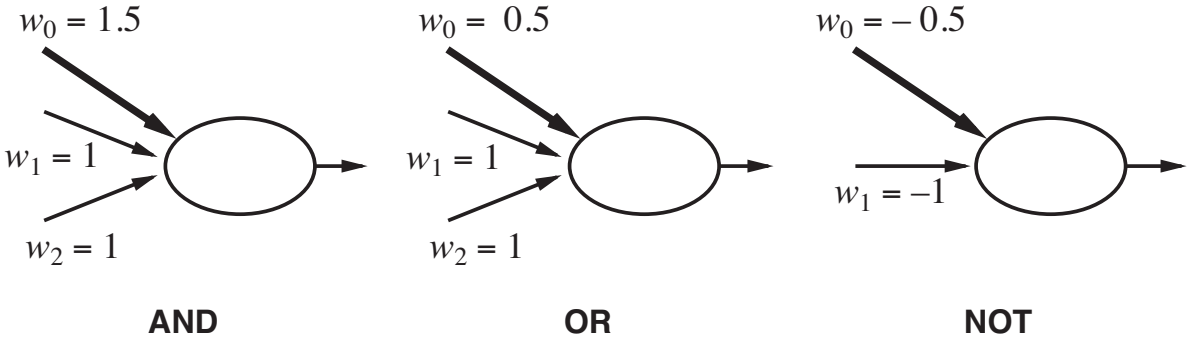
An oversimplification of real neurons, but its purpose is to develop an understanding of what networks of simple units can do

Implementing logical functions

Output is a linear (omitting g) function of the inputs, i.e., linear model (including a bias by a dummy feature)

$$y_j = \sum_i W_{i,j} x_i$$

Recall: (binary linear) classification can be viewed as a neuron



Ref. McCulloch and Pitts (1943): Boolean functions can be implemented

Question: What about **XOR**?

Perceptron: a single neuron learning

What good is a single neuron?
Idea: supervised learning (input $\mathbf{x}$, output y with target $\{0, 1\}$)

- If $t = 1$ and $z = \mathbf{W}^\top \mathbf{x} > 0$
  - then $y = 1$, so no need to change anything
- If $t = 1$ and $z < 0$
  - then $y = 0$, so we want to make z larger
  - Update

$$\mathbf{W}' \longleftarrow \mathbf{W} + \mathbf{x}$$

  - Justification

$$\begin{aligned} \mathbf{W}'^\top \mathbf{x} &= (\mathbf{W} + \mathbf{x})^\top \mathbf{x} \\ &= \mathbf{W}^\top \mathbf{x} + \mathbf{x}^\top \mathbf{x} \\ &= \mathbf{W}^\top \mathbf{x} + ||\mathbf{x}||^2 \end{aligned}$$

Perceptron learning rule

For convenience, let targets be $\{-1, 1\}$ instead of our usual $\{0, 1\}$

Perceptron Learning Rule algorithm

Repeat:

For each training case $(\mathbf{x}^{(i)}, t^{(i)})$

$z^{(i)} \leftarrow \mathbf{W}^{\mathrm{T}}\mathbf{x}^{(i)}$

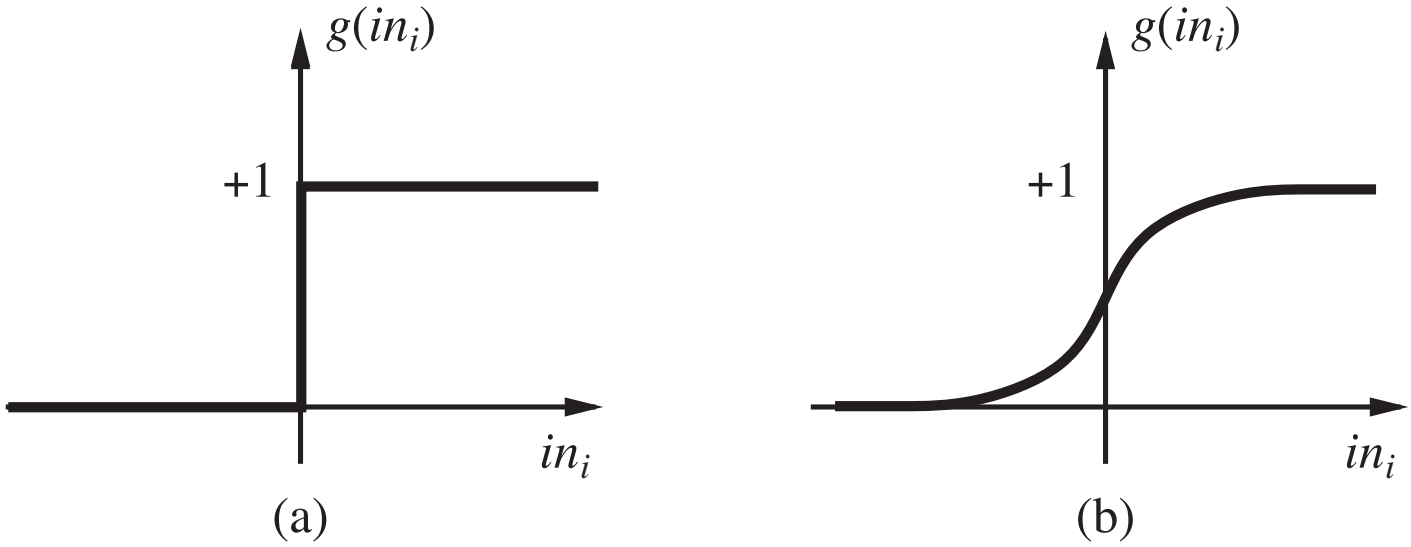
if $z^{(i)}t^{(i)} \leq 0$

$\mathbf{W} \leftarrow \mathbf{W} + t^{(i)}\mathbf{x}^{(i)}$

Stop if the weights were not updated in this epoch

- Under certain conditions, if the problem is feasible, the perceptron rule is guaranteed to find a feasible solution after a finite number of steps
- If the problem is infeasible, all bets are off

Activation functions



Perceptrons as nonlinear functions

(a) is a step function or threshold function

(b) is a sigmoid function $\sigma(x) = 1/(1 + e^{-x})$

Activation functions#

More

(c) rectified linear unit $ReLU(x) = max\{0, x\}$

– a piecewise linear function with two linear pieces

(d) softplus $\text{softplus}(x) = \log(1 + e^x)$

– a smooth version of ReLU

– the derivative of the softplus is the sigmoid

(e) tanh $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$

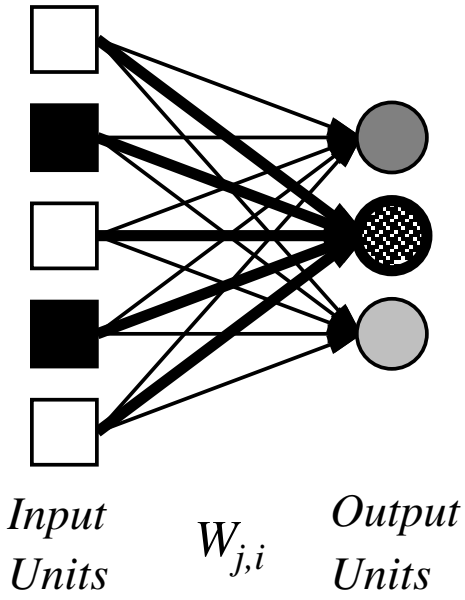
– a scaled and shifted version of the sigmoid

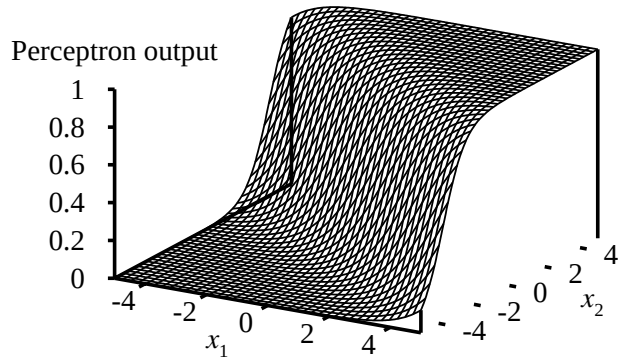
– $\tanh(x) = 2\sigma(2x) - 1)$

• All of them are monotonically nondecreasing (their derivatives g' are nonnegative)

• Changing the bias weight $W_{i,j}$ moves the threshold location (strength and sign of the connection)

Single-layer perceptrons





- Output units all operate separately — no shared weights
- Adjusting weights moves the location, orientation and steepness of the cliff

Remark: the input units do not involve any calculations and are not a layer

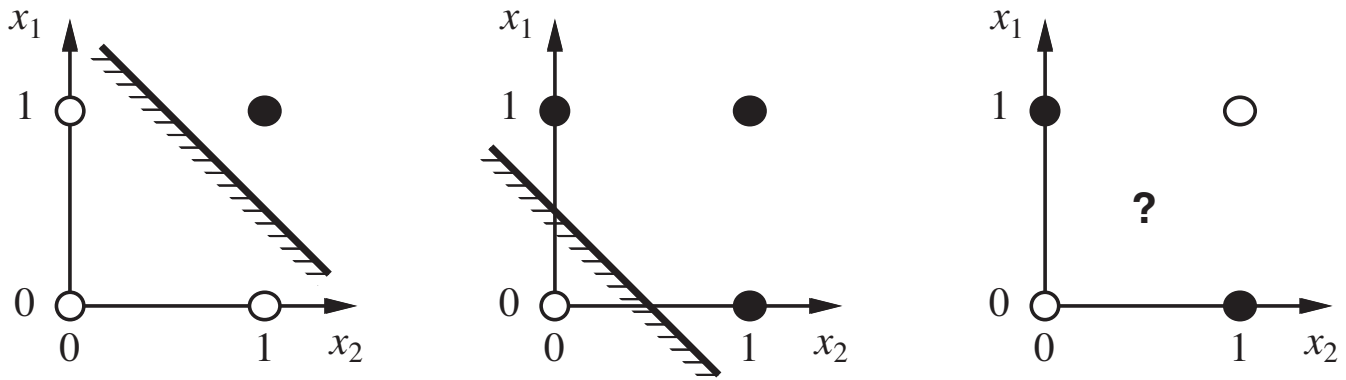
Expressiveness of perceptrons

Consider a perceptron with g = step function (Rosenblatt, 1957)

$\Rightarrow$ Can represent **AND**, **OR**, **NOT**, majority, etc.

Represents a linear separator in input space

$\sum_j W_j x_j > 0$ or $\mathbf{W} \cdot \mathbf{x} > 0$



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

But can not represent **XOR**

• Minsky & Papert (1969) pricked the neural network balloon led to the first crisis

Network structures

Feedforward networks: one direction, directed acyclic graph (DAG)

– single-layer perceptrons

– multilayer perceptrons (MLPs) — so-called deep networks

Feedforward networks implement functions, have no internal state

Recurrent (neural) networks (RNNs): feed its outputs back into its own inputs, dynamical system

– Hopfield networks have symmetric weights ($W_{i,j} = W_{j,i}$)

$g(x) = \text{sign}(x)$, $a_i = \pm 1$; holographic associative memory

– Boltzmann machines use stochastic activation functions,

$\approx$ MCMC (Markov Chain Monte Carlo) in Bayes nets

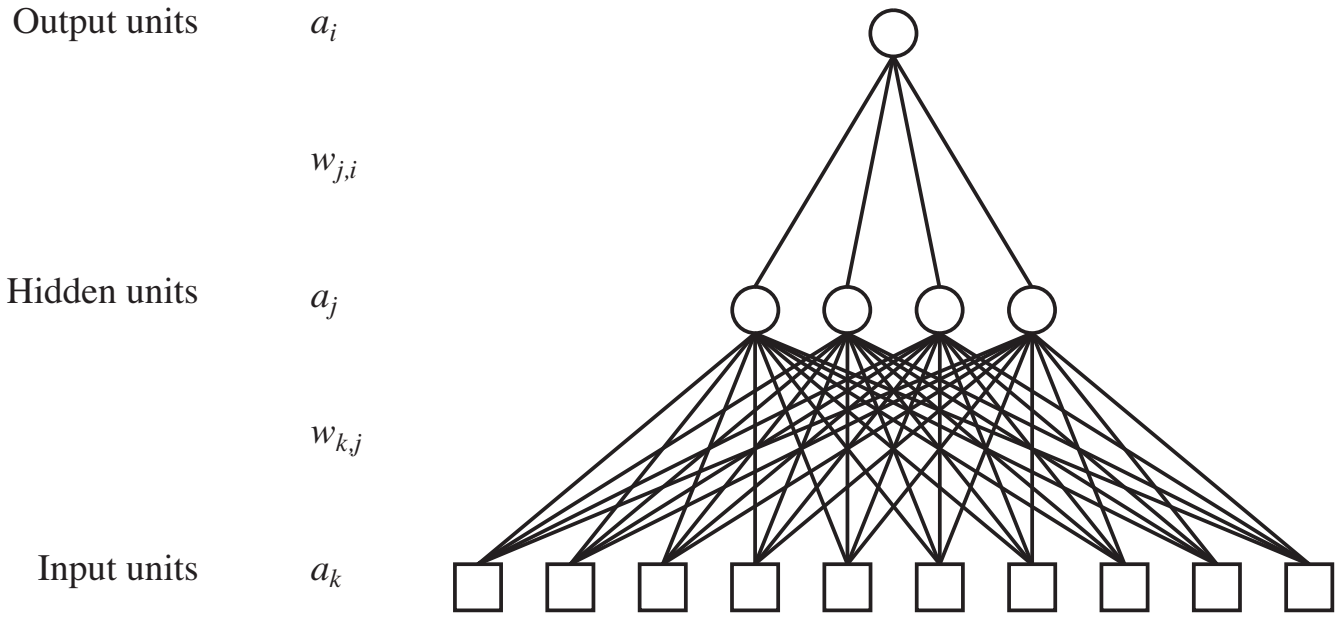
Recurrent networks have directed cycles with delays

$\Rightarrow$ has an internal state, can oscillate etc.

Neural architecture search: search (algorithms) for exploring the state space of possible network architectures

Multilayer perceptrons

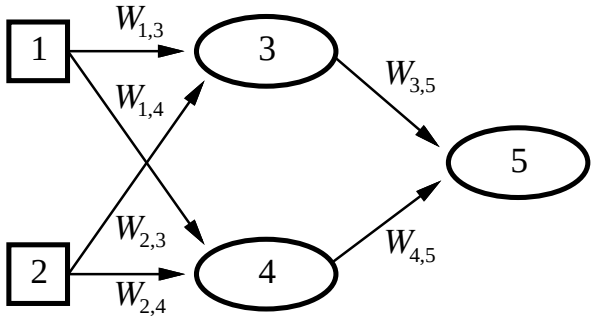
Networks (layers) are fully connected or locally connected
– numbers of hidden units typically chosen by hand



(Restaurant NN)

Fully connected feedforward network

Computation graphs for feedforward network (FFN, usually denoted as MLP)



MLPs = a parameterized family of nonlinear functions
(otherwise collapsing out the hidden to a single-layer linear model)

$$\begin{aligned} y_5 &= g(W_{3,5} \cdot x_3 + W_{4,5} \cdot x_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot x_1 + W_{2,3} \cdot x_2) + W_{4,5} \cdot g(W_{1,4} \cdot x_1 + W_{2,4} \cdot x_2)) \\ h_{\mathbf{w}} &= (\mathbf{x}) = \mathbf{g}_2\left(\mathbf{W}_2 \mathbf{g}_1\left(\mathbf{W}_1 \mathbf{x}\right)\right) \end{aligned}$$

Adjusting weights (parameters) changes the function: do **learning** this way $\Leftarrow$ supervised learning

Perceptron learning

Learn by adjusting weights to reduce error (loss) on training set
The squared error (SE) for an example with input $\mathbf{x}$ and true output y is

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2$$

Perform optimization by gradient descent (loss-min):

$$\begin{aligned}\frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g(\Sigma_{j=0}^{n} W_j x_j)\right) \\ &= -Err \times g'(in) \times x_j\end{aligned}$$

Simple weight update rule

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

E.g., +ve error $\Rightarrow$ increase network output
$\Rightarrow$ increase weights on +ve inputs, decrease on -ve inputs

Example: learning XOR

The **XOR** function: input two binary values x_1, x_2, when exactly one of these binary values is equal to 1, output returns 1; otherwise, returns 0

Training set: $\mathbf{X} = \{[0,0]^\top, [0,1]^\top, [1,0]^\top, [1,1]^\top\}$

Target function: $y = g(\mathbf{X}, \mathbf{W})$

Loss function (SE): for an example with input x and true output y is

$$E(W) = \frac{1}{4}Err^2 \equiv \frac{1}{4}\sum_{x \in \mathbf{X}}(y - h_{\mathbf{W}}(\mathbf{x}))^2$$

Suppose that $h_{\mathbf{W}}$ is choosed as a linear function

say, $h(\mathbf{X}, \mathbf{W}, b) = \mathbf{W}^\top \mathbf{X} + b$ (b is a bias)

unable to represent XOR —— Why??

Example: learning XOR

Using a MLP with one hidden layer containing two hidden units (aforesaid)

– the network has a vector of hidden units $\mathbf{h}$

Using a **nonlinear** function

$$\mathbf{h} = g(\mathbf{W}^\top \mathbf{X} + \mathbf{c})$$

where $\mathbf{c}$ is the biases, and affine transformation

– input $\mathbf{X}$ to hidden $\mathbf{h}$, vector $\mathbf{c}$

– hidden $\mathbf{h}$ to output y, scalar b

Need to use the ReLU defined by $g(z) = max\{0, z\}$ that is applied elementwise

Example: learning XOR

The complete network is specified as

$$y = g(\mathbf{X}, \mathbf{W}, \mathbf{c}, b) = \mathbf{W}_2^\top \, max\{0, \mathbf{W}_1^\top \mathbf{X} + \mathbf{c}\} + b$$

where matrix $\mathbf{W}_1$ describes the mapping from $\mathbf{X}$ to $\mathbf{h}$, and a vector $\mathbf{W}_2$ describes the mapping from $\mathbf{h}$ to y

A solution to **XOR**, let

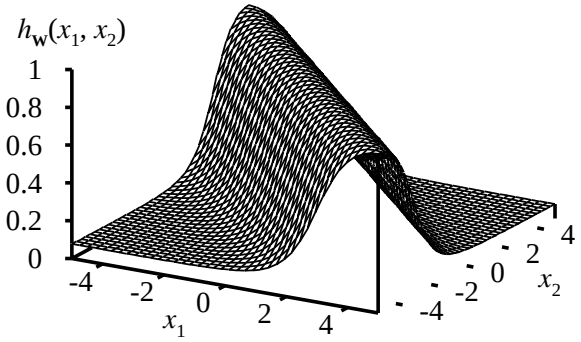
$\mathbf{W}_1 = \{[1,1]^\top, [1,1]^\top\}$
$\mathbf{W}_2 = \{[1,-2]^\top\}$
$\mathbf{c} = \{[0,-1]^\top\}$, and
$b = 0$

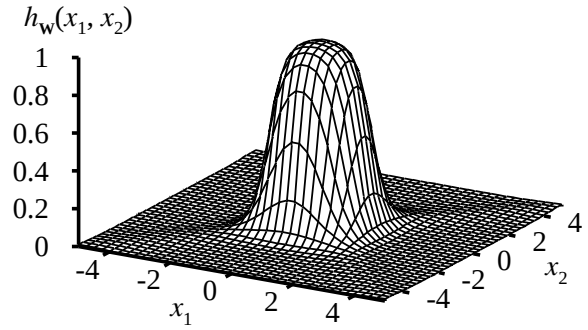
Output: $[0,1,1,0]^\top$

– The NN has obtained the correct answer for $\mathbf{X}$

Expressiveness of MLPs*

Theorem (universal approximation): All continuous functions w/ 2 layers, all functions w/ 3 layers





- Combine two opposite-facing threshold functions to make a ridge
- Combine two perpendicular ridges to make a bump
- Add bumps of various sizes and locations to fit any surface
- Proof requires exponentially many hidden units
- Hard to proof exactly which functions can(not) be represented for any particular network

Deep neural networks

DNN: using deep (n-layers, $n \geq 3$) networks to leverage large labeled datasets

– it's deep if it has more than one stage of nonlinear feature transformation

– deep vs. narrow $\Leftrightarrow$ "more time" vs. "more memory"

$\Leftarrow$ Deepness is critical, though no math proof

Let a DNN be $f_\theta(s, a)$, where

– f: the (activate) function of nonlinear transformation

– θ : the (weights) parameters

– input s: labeled data (states)

– output $a = f_\theta(s)$: actions (features)

Adjusting θ changes f: do **learning** this way (training)

Backpropagation (BP)

Output layer: same as for single-layer perceptron

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

where $\Delta_i = Err_i \times g'(in_i)$

Hidden layer: **backpropagate** the error from the output layer

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i$$

Update: rule for weights in hidden layer

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$

– The gradient of the objective function w.r.t. the input of a layer can be computed by working backward from the derivative w.r.t. the output of that layer (reverse mode differentiation)
– Most neuroscientists deny that backpropagation occurs in the brain

BP derivation

The SE on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2$$

where the sum is over the nodes in the output layer

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i)g'(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i)g'(in_i)\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i}a_j\right) \\
&= -(y_i - a_i)g'(in_i)a_j = -a_j\Delta_i
\end{aligned}$$

– Objective (loss) functions: SE, likelihood, cross-entropy etc.
– Gradients can be computed by the automatic differentiation
– SGD commonly used in training

BP derivation#

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j
\end{aligned}$$

BP learning algorithm

def BP-LEARNING(*examples*, *network*)
  inputs: *examples*, a set of examples, each /w in/output vectors X and Y
  local variables: Δ, a vector of errors, indexed by network node

  repeat
    for each weigh $w_{i,j}$ in *networks* **do**
      $w_{i,j} \leftarrow$ a small random number
    for each example $(X,\ Y)$ **in** *examples* **do**
      for each node i in the input layer **do**
        $a_i \leftarrow x_i$
      for $l = 2$ **to** L **do**
        for each node j in layer l **do**
          $in_j \leftarrow \Sigma_i\ w_{i,j}\, a_i$
          $a_j \leftarrow g(in_j)$

BP learning algorithm

for each node j in the output layer **do**
  $\Sigma[j] \leftarrow g'(in_j) \times (y_j - a_j)$
for $l = L - 1$ **to** 1 **do**
  for each node i in the layer l **do**
    $\Delta[i] \leftarrow g'(in_i) \Sigma_j \, w_{i,j} \Delta[j]$
  for each weight $w_{i,j}$ in $network$ **do**
    $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$
until some stopping criterion is satisfied
return $nerwork$

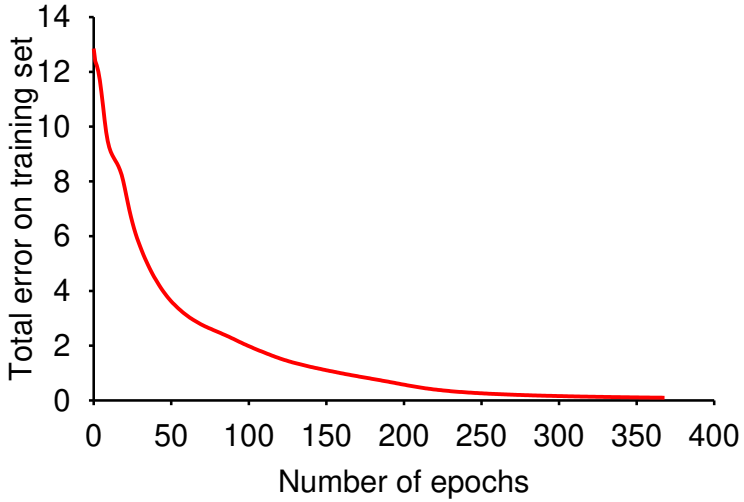
End-to-end learning: the system is trained based on BP in an end-to-end fashion from input/output pairs

need only about how the overall system should be structured; no need to know exactly what should do its inputs/outputs

BP learning

At each epoch, sum gradient updates for all examples and apply

Training curve for 100 restaurant examples: finds exact fit



DNNs are quite good for complex pattern recognition tasks,
but resulting hypotheses cannot be **interpreted** (black box method)
Problems: gradient vanishing (or exploding), slow convergence, local minima

Convolutional neural networks

CNNs: DNNs that use convolution in place of general matrix multiplication (in at least one of their layers)

- locally connected networks

  spatial invariance at the local region, cutting down the number of weights, e.g. for an image, if each local region has $l \ll n$ pixels, then there will be weights $ln \ll n^2$ in all

- for processing data that has a known grid-like topology

  e.g., time-series data, as a 1D grid taking samples at regular time intervals; image data, as a 2-D grid of pixels

- any NN algorithm that works with matrix multiplication and does not depend on specific properties of the matrix structure should work with convolution

Convolutional function

$$
\begin{aligned}
s(t) &= (x * w)(t) \\
&= \int x(a)w(t-a)da \\
&= \sum_{a=-\infty}^{\infty} x(a)w(t-a) \\
&= \boldsymbol{XW} \text{ (dot product of tensors)}
\end{aligned}
$$

- x: input
- w: kernel (filter)

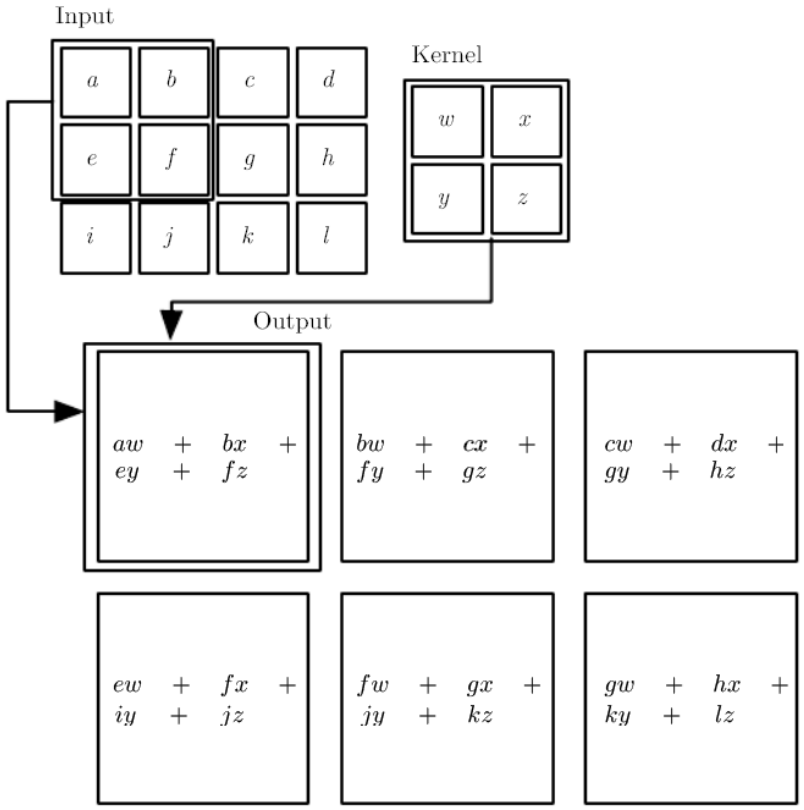
– valid probability density function or the output will not be a weighted average

– needs to be 0 for all negative arguments, or will look into the future (which is presumably beyond the capabilities)

- s: feature map

Smoothed estimate of the input data, weighted average (more recent measurements are more relevant)

Example: convolutional operation



A 2D convolution operation with a kernel of size $l = 2 \times 2$ and a stride $s = 0$

Convolutional operation

Convolution with a single kernel can extract only one kind of feature
There may be d kernels, with a stride of 1, the output will be d times larger

a n-dimensional input matrix becomes a $n+1$-dimensional matrix of hidden units, where the $n+1$th dimension is of size d

each channel carries information from one feature extracted by a kernel

Remark*
Don't implement by a program with nested loops
Can be formulated as a single matrix (tensor) operation

– matrix is more efficient than loops, e.g., Numpy (Python) by using GPU/TPU

– the weight matrix is spare (mostly zeroes), and convolution is a linear matrix operation that gradient descent that can be applied easily and effectively

Pooling#

A pooling layer in an NN summarizes a set of adjacent units from the preceding layer with a single value (fixed rather than learned, no activation function) $\Rightarrow$ downsampling

- Average-pooling: the average value of its inputs
- Max-pooling: the maximum value of its inputs

Residual networks#

Need to build very deep networks (hundreds of layers) that avoid the vanishing gradients

Idea: a layer i should *perturb* the representation from the previous layer $i-1$ rather than *replace* it entirely

$$\mathbf{z}^{(i)} = f\left(\mathbf{z}^{(i-1)}\right) = \mathbf{g}^{(i)}\left(\mathbf{W}^{(i)}\mathbf{z}^{(i-1)}\right)$$

i-layer completely replaces the representation from $i-1$-layer

$$\mathbf{z}^{(i)} = \mathbf{g}_r^{(i)}\left(\mathbf{z}^{(i-1)} + f\left(\mathbf{z}^{(i-1)}\right)\right)$$

$\mathbf{g}_r$ – the activation functions for the residual layer

Residual networks

f – residual, perturbing the default behavior of passing i-layer through to $i-1$-layer

typically an NN with one nonlinear layer combined with one linear layer

$$f(\mathbf{z}) = \mathbf{V}\mathbf{g}(\mathbf{W}\mathbf{z})$$

$\mathbf{V}$, $\mathbf{W}$ – learned weight matrices with the bias weights added

If the learned perturbation is small, the next layer is close to being a copy of the previous layer

Set $\mathbf{V} = \mathbf{0}$ to disable a particular layer, the f disappears

$$\mathbf{z}^{(i)} = \mathbf{g}_r\left(\mathbf{z}^{(i-1)}\right)$$

Residual networks

Let the activation function be ReLU

$$\begin{aligned}\mathbf{z}^{(i)} &= \mathbf{g}_r\left(\mathbf{z}^{(i-1)}\right) = \mathrm{ReLU}\left(\mathbf{z}^{(i-1)}\right) \\ &= \mathrm{ReLU}\left(\mathrm{ReLU}\left(\mathbf{in}^{(i-1)}\right)\right) = \mathrm{ReLU}\left(\mathbf{in}^{(i-1)}\right) \\ &= \mathbf{z}^{(i-1)}\end{aligned}$$

A layer with zero weights simply passes its inputs through with no change

To avoid catastrophic failure of the propagation for bad choices of the parameters, residual networks propagate information by the design

Batch normalization#

Batch normalization (BN): improves the rate of convergence of SGD by rescaling the values generated at the internal layers of the network from the examples within each minibatch

Consider a node z: the values of z for the m examples in a minibatch are $z_1, \cdots, z_m$

BN: replacing each z_i with a new quantity $\hat{z}_i$

$$\hat{z}_i = \gamma \frac{z_i - \mu}{\sqrt{\epsilon + \sigma^2}} + \beta$$

– μ: the mean value of z across the minibatch
– σ: the standard deviation of $z_1, \cdots, z_m$
– ϵ: a small constant added to prevent division by zero
– γ, β: learned parameters

Dropout#

Dropout: at each step of training, applies one step of BP learning by deactivating a randomly chosen subset of the units

– reducing the testing error (better generalization) of a network at the cost of making it harder to fit the training set

Consider using SGD with minibatch size m

for each minibatch, dropout applies to every node (in the hidden layers) of the network

with probability p (say $p = 0.5$ for hidden $p = 0.8$ for input), the unit output is multiplied by a factor of $1/p$; otherwise, the unit output is fixed at zero

– Produced a thinned network (about half original units)

– At test time, the model is run with no dropout

Input encoding

Input encoding: data $\mathbf{x}$

- Factored data: encoding just as the attribute values (Boolean, numeric or real values)
- Images: RGB, representing color images with three numbers per pixel (principle of trichromacy)

– an RGB image of size $X \times Y$ pixels can be thought of as $3XY$ integer-valued attributes (in the range $\{0, \cdots, 255\}$)

Output layer

Output layer: data $\mathbf{x}$

- Encoding the raw data values into actual values (the same as the input encoding)
- Loss function: prediction $\hat{y}$
  - SE (squared-error), MLE (negative log-likelihood)
  - CE (cross-entropy): $H(P,Q) = \mathbf{E}_{\mathbf{z}\sim P(\mathbf{z})}[\log Q(\mathbf{z})] = \int P(\mathbf{z}) \log Q(\mathbf{z}) d\mathbf{z}$

- Softmax layer: needing to minimize MLE (CE) by interpreting the output as a probability summing to 1

  outputs a vector of d values, given a vector of input values in $= \langle in_1, \cdots, in_d \rangle$

  the k-th element of that output vector is given by softmax function

$$\text{softmax}(\text{ in })_k = \frac{e^{in_k}}{\sum_{k'=1}^{d} e^{in_{k'}}}$$

Hidden layers

End-end training hypothesis: for why DNN work well is that the compositional end-to-end transformation that maps from input to output

– learning at each layer a meaningful representation for the input

– internal nodes only used activation functions such as ReLU

Write $f_\theta(h_t) = f_\theta(h_t, \boldsymbol{x}_t, \boldsymbol{y}_t)$ to indicate the t-th hidden layer of f_θ with the deepness T ($t \in \mathbb{N}$)

– $(h_{t-1}(\boldsymbol{x}_1), \cdots, h_{t-1}(\boldsymbol{x}_{t-1}))$ are the input vectors from the previous layer

– $\boldsymbol{y}_t = h_t(\boldsymbol{x}_t)$ is the output vector

— at every hidden layer, updating the previous hidden layer h_{t-2}, which summarizes $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{t-2})$, with a new representation $\boldsymbol{x}_{t-1}$, resulting in current hidden layer h_{t-1}

Example: Drawing a CNN picture

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# Create figure and axes
fig, ax = plt.subplots(figsize=(8, 6))

# Define layers and their sizes
layers = ['Input', 'Convolutional', 'Pooling', 'Fully Connected', 'Output']
sizes = [1, 3, 2, 1, 1]

# Plot each layer
x_offset = 0
for layer, size in zip(layers, sizes):
    rect = patches.Rectangle((x_offset, 0), size, 1, linewidth=2, edgecolor='b', facecolor='none')
    ax.add_patch(rect)
    ax.text(x_offset + size / 2, 0.5, layer, ha='center', va='center')
    x_offset += size

# Set axis limits and hide axis
ax.set_xlim(0, sum(sizes))
ax.set_ylim(0, 1)
ax.axis('off')

# Show plot
plt.show()
```

Recurrent neural networks

RNNs: DNNs for processing sequential data

– process a sequence of values $x^{(1)}, ..., x^{(\tau)}$

e.g., natural language processing (speech recognition, machine translation etc.)

– can scale to much longer sequences than would be practical for networks without sequence-based specialization

– can also process sequences of variable length

Learning: predicting the future from the past

Recurrence

Classical form of a dynamical system

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}) \tag{1}$$

where $\boldsymbol{s}^{(t)}$ is the state

Recurrence: the definition of $\boldsymbol{s}$ at time t refers back to the same definition at time $t-1$

Dynamical system driven by an external signal $\boldsymbol{x}^{(t)}$

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) \tag{2}$$

$\boldsymbol{h}$ (except for input/output): hidden units, and the state contains information about the whole past sequence

Any function involving recurrence can be considered an RNN

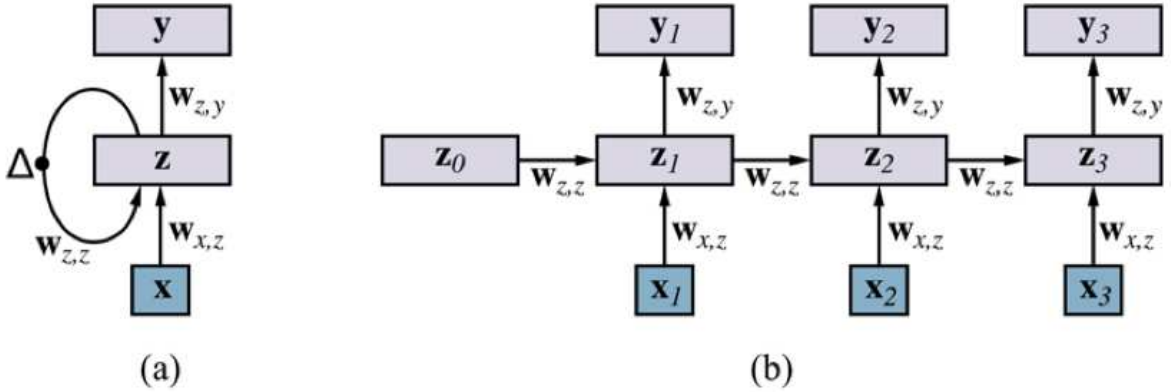
RNN learns to use $\boldsymbol{h}(t)$ as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t

RNN

$\mathbf{z}_t = f_{\mathbf{w}}(\mathbf{z}_{t-1}, \mathbf{x}_t) = \mathbf{g}_z(\mathbf{W}_{z,z}\mathbf{z}_{t-1} + \mathbf{W}_{x,z}\mathbf{x}_t) \equiv \mathbf{g}_z(\mathbf{in}_{z,t})$

$\hat{\mathbf{y}}_t = \mathbf{g}_y(\mathbf{W}_{z,y}\mathbf{z}_t) \equiv \mathbf{g}_y(\mathbf{in}_{y,t})$

– input $\mathbf{x}$, hidden $\mathbf{z}$, output $\mathbf{y}$



(a) An RNN hidden layer $\mathbf{z}$ with recurrent connections
(b) Unfolded computational graph

RNN (Markov) assumption: the hidden state $\mathbf{z}_t$ suffices to capture the information from all previous inputs

BP through time#

$$\frac{\partial L}{\partial w_{z,z}} = \frac{\partial}{\partial w_{z,z}} \sum_{t=1}^{T} (y_t - \hat{y}_t)^2 = \sum_{t=1}^{T} -2 (y_t - \hat{y}_t) \frac{\partial \hat{y}_t}{\partial w_{z,z}}$$

$$= \sum_{t=1}^{T} -2 (y_t - \hat{y}_t) \frac{\partial}{\partial w_{z,z}} g_y (in_{y,t})$$

$$= \sum_{t=1}^{T} -2 (y_t - \hat{y}_t) g'_y (in_{y,t}) \frac{\partial}{\partial w_{z,z}} in_{y,t}$$

$$= \sum_{t=1}^{T} -2 (y_t - \hat{y}_t) g'_y (in_{y,t}) \frac{\partial}{\partial w_{z,z}} (w_{z,y} z_t + w_{0,y})$$

$$= \sum_{t=1}^{T} -2 (y_t - \hat{y}_t) g'_y (in_{y,t}) w_{z,y} \frac{\partial z_t}{\partial w_{z,z}}$$

BP through time#

$$\begin{aligned}\frac{\partial z_t}{\partial w_{z,z}} &= \frac{\partial}{\partial w_{z,z}} g_z\left(in_{z,t}\right) = g_z'\left(in_{z,t}\right)\frac{\partial}{\partial w_{z,z}} in_{z,t} \\ &= g_z'\left(in_{z,t}\right)\frac{\partial}{\partial w_{z,z}}\left(w_{z,z}z_{t-1} + w_{x,z}x_t + w_{0,z}\right) \\ &= g_z'\left(in_{z,t}\right)\left(z_{t-1} + w_{z,z}\frac{\partial z_{t-1}}{\partial w_{z,z}}\right)\end{aligned}$$

where the last line uses the rule for derivatives of products

$\partial(uv)/\partial x = v\partial u/\partial x + u\partial v/\partial x$

Problems: vanishing gradient if $w_{z,z} < 1$, exploding gradient if $w_{z,z} > 1$

RNN models*

RNN architectures are designed with the goal of enabling information to be preserved over many time steps

- Long short-term memory (LSTM): a variant of basic RNN
- Gated recurrent unit (GRU): a variant of LSTM

Theorem: Any function computable by a Turing machine can be computed by an RNN of a finite size

Deep models*

- Autoregressive model (AR): each element x_i of the data vector x is predicted based on other elements of the vector

  e.g., time series data, the order k predicts x_t given $x_{t-k}, \cdots, x_{t-1}$

  – conditional distribution $P\left(x_t \mid x_{t-k}, \ldots, x_{t-1}\right)$

Autoencoder (AE): two parts

– encoder f: mapping from x to a representation $\hat{z}$

– decoder g: mapping from a representation $\hat{\mathbf{z}}$ to observed data $\mathbf{x}$

– the model is trained so that $\mathbf{x} \approx g(f(\mathbf{x}))$ (the encoding process is roughly inverted by the decoding process)

Variational autoencoder (VAE): using variational methods in AE

Deep models*

- Generative adversarial network (GAN): a pair

  – generator: mapping values from z to x to produce samples from the distribution $P_{\mathrm{w}}(\mathbf{x})$

  say, sampling $\mathbf{z}$ from a Gaussian and then passes it through a deep network h_{w} to obtain x

  – discriminator: a classifier trained to classify inputs x as real (drawn from the training set) or fake (created by the generator)

Deep learning#

Deep learning = representations (features) learning

– introducing representations that are expressed in terms of other simpler representations

– data ⇒ representation (learning automatically)

Pattern recognition: fixed/handcrafted features extractor

→ features extractor → (mid-level features) → trainable classifier

Deep learning: representations are hierarchical and trained

→ low-level features → mid-level features → high-level features → trainable classifier →

– the entire machine is trainable

E.g., Image: pixel → edge → texton → motif → part → object

Speech: sample → · · · → phone → phoneme → word

Text: character → word → word groups → clause → sentence → story

Example: Games as perception*

Alpha0 Go Implementation

– raw board representation: $19 \times 19 \times 17$

historic position $s_t = [X_t, Y_t, X_{t-1}, Y_{t-1}, \cdots, X_{t-7}, Y_{t-7}, C]$

Treat as two-dimensions images

– CNN has a long history in computer Go by self-play reinforcement learning (Schraudolph N *et al.*, 1994)

Alpha0 Go: $EvalFn \Leftarrow$ stochastic simulation $\Leftarrow$ (deep) learning

Deep vs shallow*

Deep and narrow vs. shallow and wide $\Leftrightarrow$ "more time" vs. "more memory"

– algorithm vs. look-up table

– few functions can be computed in 2 steps without an exponentially large lookup table

– using more than 2 steps can reduce the "memory" by an exponential factor

All major deep learning frameworks use modules

– Torch7, Theano, TensorFlow etc.

Any architecture (connection graph) is permissible

Deep learning fantasist*

- Idealist: some people hope that a **single** deep learning algorithm to study many or even all of these application areas simultaneously
  – finally, deep learning = AI = principle of intelligence

- Brain: deep learning is more likely to cite the brain as an influence, but should not view deep learning as an attempt to simulate the brain
  – today, neuroscience is regarded as an important source of inspiration for deep learning, but it is no longer the predominant guide for the field
    – differ from Artificial Brain

- Math: deep learning draws inspiration from especially applied math
  – maybe deep learning replaces applied math

Deep learning ⊂ machine learning ⊂ AI

Deep learning faces some big challenges
- formulating unsupervised deep learning
- uninterpreted black-box
- how to do reasoning etc.

Reading *LeCun&Bengio&Hinton, Deep learning, Nature 521, 436-444, 2015*
www.nature.com/nature/journal/v521/n7553/full/nature14539.html

Ref. Goodfellow&Bengio&Courville, Deep learning, MIT press
www.deeplearningbook.org

Deep Learning Papers Reading Roadmap
github.com/songrotek/Deep-Learning-Papers-Reading-Roadmap

Mathematical analysis of deep learning*

Questions have not been answered within classical learning theory and mathematics — not explainable

- The outstanding generalization of overparametrized neural networks
- The role of depth in architectures
- The absence of the curse of dimensionality
- The successful optimization performance despite the non-convexity of the problem
- What features are learned, why deep architectures perform exceptionally well
- Which fine aspects of architecture affect the behavior of a learning task in which way

A newly emerging field for math

Example: Alpha0*

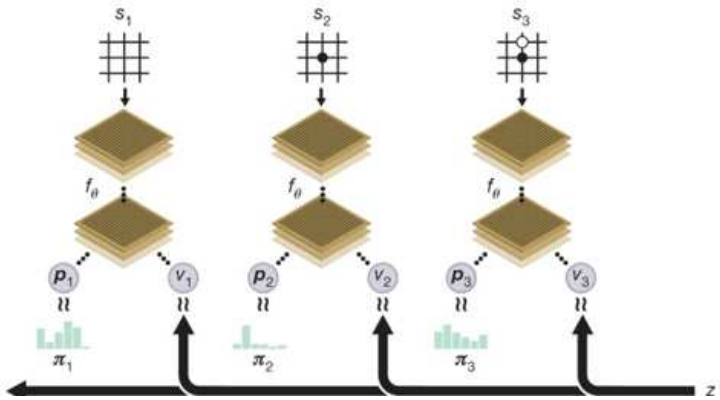
Recall the design of Alpha0 algorithm

1. combine deep learning in an MCTS algorithm
   – a single DNN for both
     police for breadth pruning, and
     value for depth pruning
2. in each position, an MCTS search is executed guided by the DNN
   with data by self-play reinforcement learning
   without human knowledge beyond the game rules
3. asynchronous multi-threaded search that executes simulations
   on CPUs, and computes DNN in parallel on GPUs

Example: Alpha0 deep learning

A DNN f_θ with parameters θ

a. input: raw board representation of the position and its history s_t, π_t, z_t (samples from SELFPLAY)
b. passing it through many convolutional layers (CNN) with θ



Example: Alpha0 neural network training

Deep learning: f_θ training

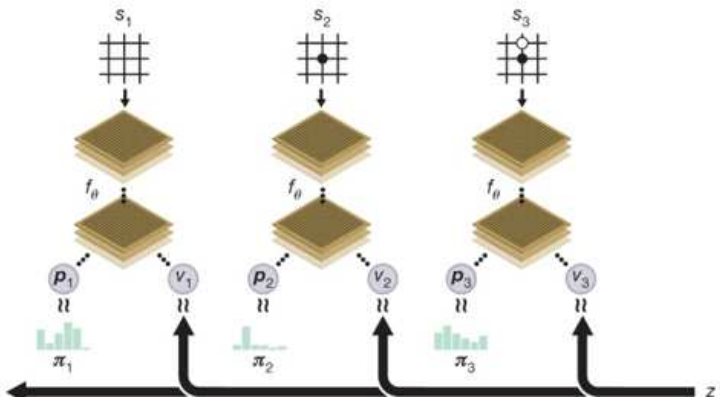
c. updating θ (for best (π, z))

– to maximize the similarity of $\mathbf{p}_t$ to the search probabilities π_t

– to minimize the error between the predicted winner v_t and the game winner z

$$(\mathbf{p}, v) = f_\theta(s), l = (z - v)^2 - \pi^T log\mathbf{p} + c||\theta||^2$$

where c is a parameter controlling the level of $L2$ weight regularization (to prevent overfitting)

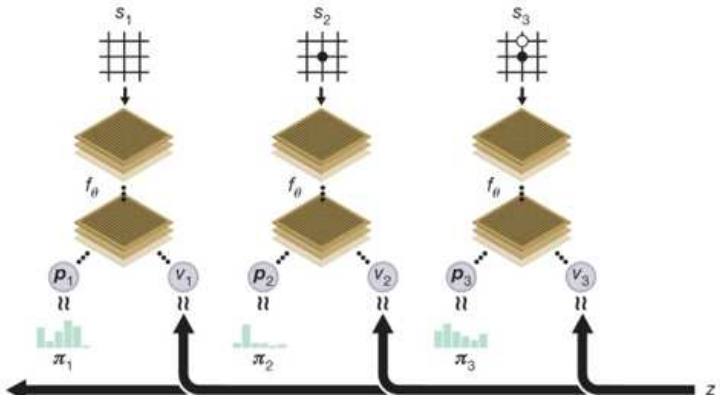


Example: Alpha0 deep learning

d. output $(\mathbf{p}, v) = f_\theta(s)$: move probabilities and a value

— vector $\mathbf{p}$: the probability of selecting each move a (including pass), $p_a = P(a \mid s)$

– a scalar evaluation v: the probability of the current player winning from position s



(MCTS outputs probabilities π of playing each move from $\mathbf{p}$)

Example: Alpha0 deep learning pseudocode

```
def DNN(s_t)
    inputs: f_θ, CNN with parameters θ
    // say, 1 convolutional block + 39 residual blocks
    // policy head (2 layers) + value head (3 layers)
    s_t: historic data, initially random

    while within computational budget do
        for each s_t do
            data ← SELFPLAY(s_t,π_t,z_t)
            (p_t,v_t) ← CNN(f_θ(data))
            π_t ← MCTS(f_θ(s_t))
            s_t ← UPDATE(a_t,π_t)
    return θ(BESTPARAMETERS(f))
```

Reinforcement learning

Reinforcement learning (RL): learn what to do in the absence of labeled examples of what to do

– learn from success (reward) and failure (punishment)

RL vs. planning and supervised/unsupervised learning

– given model of how decisions impact the world (replanning)

– rewards as labels, not correct labels, or no labels

Imitation learning: Learns from the experience of others, assumed input demos of good policies

– reduces RL to supervised learning

In many domains, reinforcement learning is the only feasible way to train a program to perform certain tasks

RL agents

Recall agent architectures

- Utility-based agent: learn a utility function
- Q-learning agent: action-utility function, giving the expected utility of taking a given action in a given state
- Reflex agent: learn a policy that maps directly from states to actions

Components of an RL agent

– Model: the world changes in response to the action

– Policy: function mapping agent's states to action

– Value (utility): future rewards from being in a state and/or action when following a particular policy

Exploration and exploitation

Recall

- Exploration: trying new things that enable the agent to make better decisions in the future
- Exploitation: choosing actions that are expected to yield good rewards given past experience

Often there may be an exploration-exploitation tradeoff

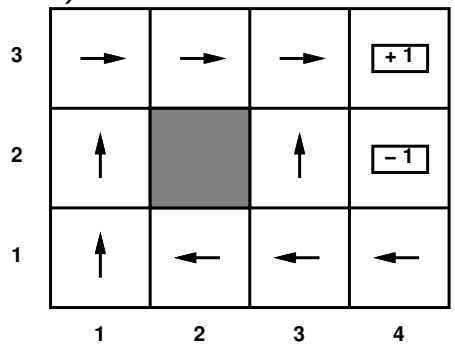
Passive and active RL

In passive learning, the agent's policy π is fixed

– in state s, execute the action $\pi(s)$

– to learn the utility function $U^{\pi}(s)$

Recall: MDP (Markov decision process) $\Rightarrow$ MRP=MDP+rewards



– to find an optimal policy $\pi(s)$

But passive learning agent does not know the transition model $P(s'|s,a)$ and the reward function $R(s)$

Passive and active RL

```
def PASSIVE-RL-AGENT(e)
    persistent: U, a table of utility estimates
                N, a table of frequencies for states
                M, a table of transition probabilities from state to state
                percepts, a percept sequence (initially empty)
    add e to percepts
    increment N[STATE[e]]
    U ← UPDATE(U, e, percepts, M, N)
    if TERMINAL?[e] then percepts ← the empty sequence
    return the action Observe
```

An active learning agent decides what actions to take

Learning action-utility functions instead of learning utilities

— Q-learning

Q-learning

Q-functioln $Q(s,a)$: value of doing action a in state s

Q-value: $U(s) = max_a Q(s,a)$

– not need a model of $P(s'|s,a)$ (model-free)

– representation by a lookup table or function approximation

– – with function approximation, reduce to supervised learning

(learning a model for an observable environment is supervised learning, because the next percept gives the outcome state)

– – – any supervised learning methods can be used

Policy search

Idea: iterating the policy as long as its performance improves, then stop

a policy π can be represented by a collection of parameterized Q-functions (one for each action), and take the action with the highest predicted value

$$\pi(s) = max_a \hat{Q}_\theta(s, a)$$

– Q-function could be a linear function of the parameters, or
  a nonlinear function, such as a neural network
  thus, policy search results in a process that learns Q-functions

Policy interation

PI algorithm

1. Initially $\pi_0(s)$ randomly for all s
2. While $|\pi_i - \pi_{i-1}| > 0$ (L_1 norm measures if the policy changed)
   - Policy evaluation by computing Q^{π_i}
   - Policy improvement by $\pi(s) = max_a \hat{Q}_\theta(s,a)$
     define $\pi_{i+1}(s) = argmax_a \hat{Q}^{\pi_i}(s,a), \forall s$

Optimization

– policy gradient by $\bigtriangledown_\theta \pi(\theta)$, or by score function as $\bigtriangledown_\theta \log \pi(\theta)(s,a)$

– empirical gradient (gradient-free) by hill climbing, genetic algorithms etc.

Deep reinforcement learning

DRL: use deep neural networks to represent

– value function

– policy model

Optimize loss function by SGD (stochastic gradient descent)

Policy evaluation Q^{π_i} by function approximation, using deep learning

– nonlinear function (differential)

– advantages of deep learning

– – scale up to making decisions in really large domains, etc.

Deep Q-learning#

DQN (deep Q-networks): represent value function by a deep neural network (Q-network) with weights θ

- $\hat{Q}(s,a,\theta) \approx Q(s,a)$
- Minimize MSE loss by SGD

Actor-critic algorithms#

- Actor: the policy
- Critic: value function (Q-funciton)
- Reduce variance of policy gradient

Policy evaluation

– fitting value function to policy

Design

– One network (with two heads) or two networks

Self-play#

A RL agent learns to improve its performance by playing against itself

Self-play algorithm

1. **Initialization**: Some basic policy
2. **Self-Play Iterations**:

   2.1 the agent plays the game against itself using its current policy

   2.2 the agent learns from its experiences, typically using techniques like Q-learning or policy gradient methods

   2.3 the agent updates its policy to improve its performance based on the outcomes of these games
3. **Policy Improvement**:

   3.1 after a certain number of iterations or when a performance criterion is met, the agent evaluates its current policy

   3.2 the agent may then update its policy based on the learned experiences, aiming to improve its performance in future games
4. **Repeat**: The process iterates, learning from its own experiences

Example: Alpha0 self-play training*

Reinforcement learning: Play a game $s_1, \cdots, s_T$ against itself

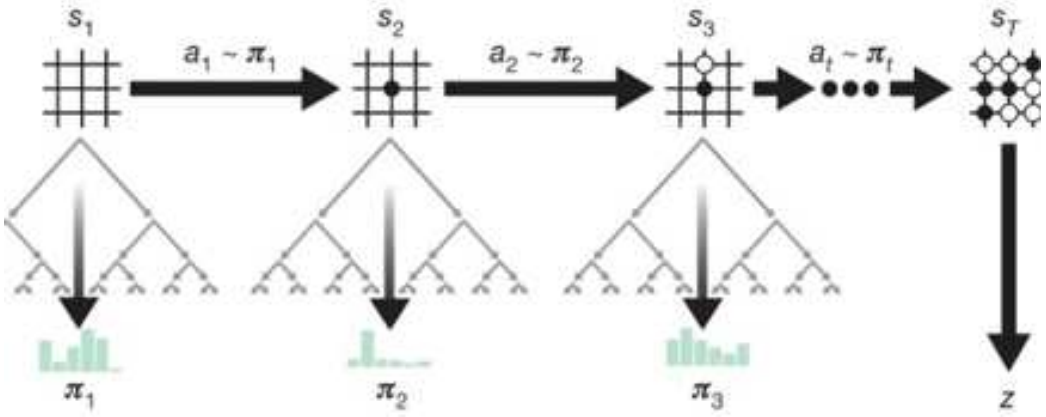
a. input: current position with search probability π (α_θ)

b. in each s_t, an MCTS α_θ is executed using the latest DNN f_θ

c. moves are selected according to the search probabilities computed by the MCTS, $a_t \sim \pi_t$

d. the terminal position s_T is scored according to the rules of the game to compute the game-winner z

e. output: sample data of a game



Example: Alpha0 self-play training pseudocode

```
def SelfPlay(state, π)
    inputs: game, s_1, ···, s_T
    create root node with state s, initially random play
    while within computational budget do
        for each s_t do
            (a_t, π_t) ← Mcts(s_t, f_θ)
            data ← DataMaking(game)
            z ← Win(s_T)
    return z(Winner(data))  //training data
```

Statistical learning$^{+\#}$

Learning as a form of uncertain reasoning from observations

Learning a probabilistic model (say, Bayesian networks)

given data that are assumed to be generated from that model, called density estimation

Bayesian learning: updating of a probability distribution over the hypothesis space (all the hypotheses)

learning is reduced to probabilistic inference

H is the hypothesis variable with values $\ldots h_i \ldots$, prior $\mathbf{P}(H)$

The jth observation d_j gives the outcome of random variable $D_j \in D$

training data $\mathbf{d} = d_1, \ldots, d_N$

Bayesian learning

Given the data so far, each hypothesis has a posterior probability by Bayes' rule

$$P(h_i|\mathbf{d}) = \frac{P(\mathbf{d}|h_i)P(h_i)}{P(\mathbf{d})} = \alpha P(\mathbf{d}|h_i)P(h_i) \propto P(\mathbf{d}|h_i)P(h_i)$$

$P(\mathbf{d}|h_i)$ is called the likelihood

$$Posterior = \frac{likelihood \times prior}{Evidence}$$

Predictions about an unknown quantity X use a likelihood-weighted average over the hypotheses

$$\mathbf{P}(X|\mathbf{d}) = \sum_i \mathbf{P}(X|\mathbf{d}, h_i)P(h_i|\mathbf{d}) = \sum_i \mathbf{P}(X|h_i)P(h_i|\mathbf{d})$$

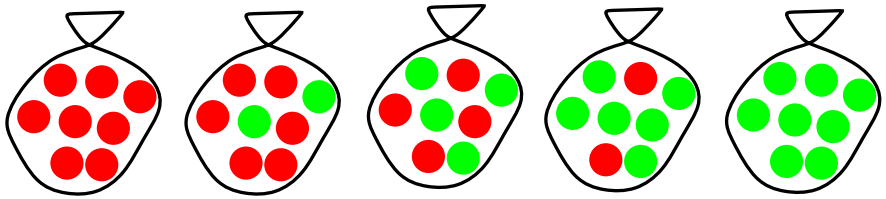
assumed that each hypothesis determines a distribution over X

No need to pick one best-guess hypothesis

Example#

Suppose there are five kinds of bags of candies with prior distribution

10% are h_1: 100% cherry candies
20% are h_2: 75% cherry candies + 25% lime candies
40% are h_3: 50% cherry candies + 50% lime candies
20% are h_4: 25% cherry candies + 75% lime candies
10% are h_5: 100% lime candies

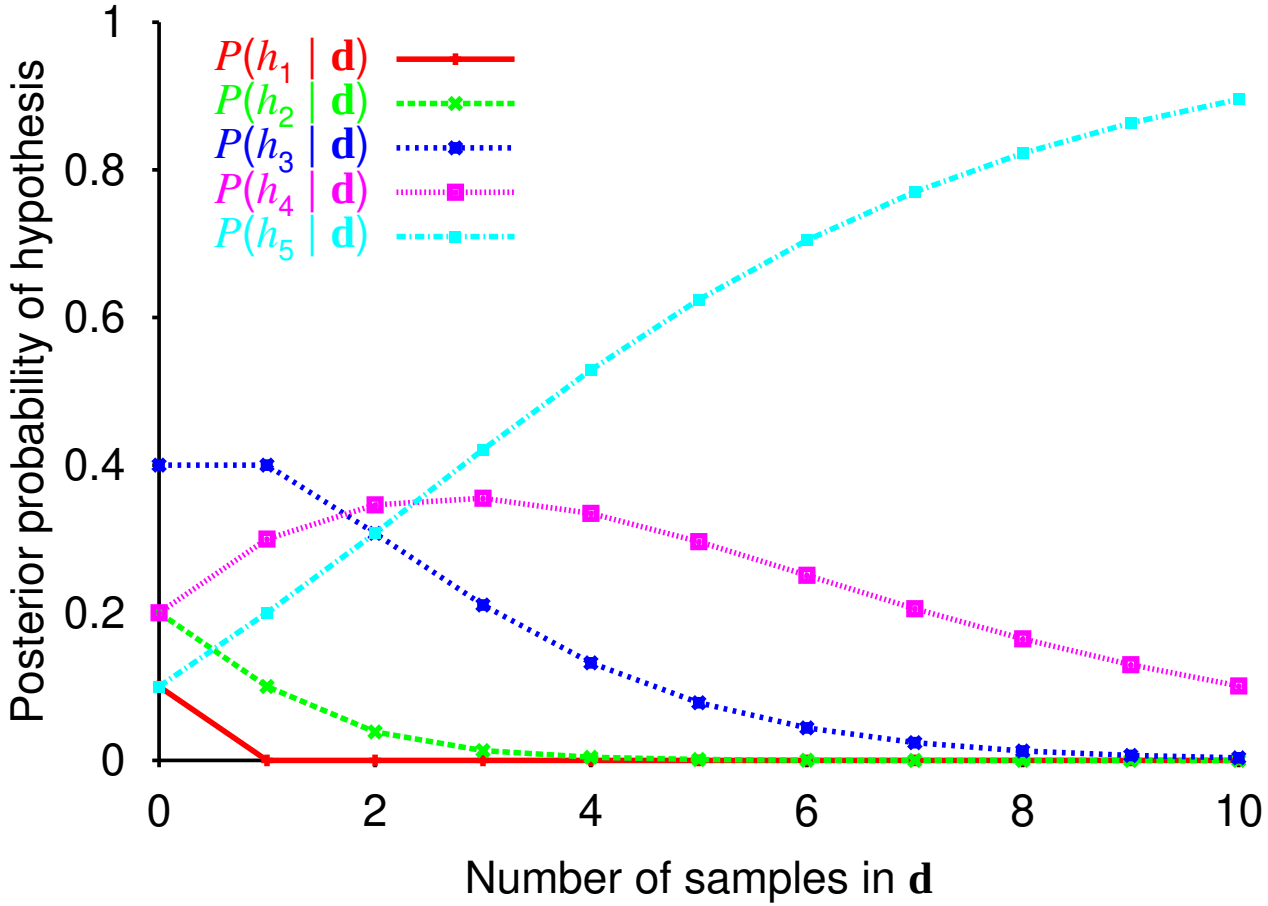


Then we observe candies drawn from some bag: 

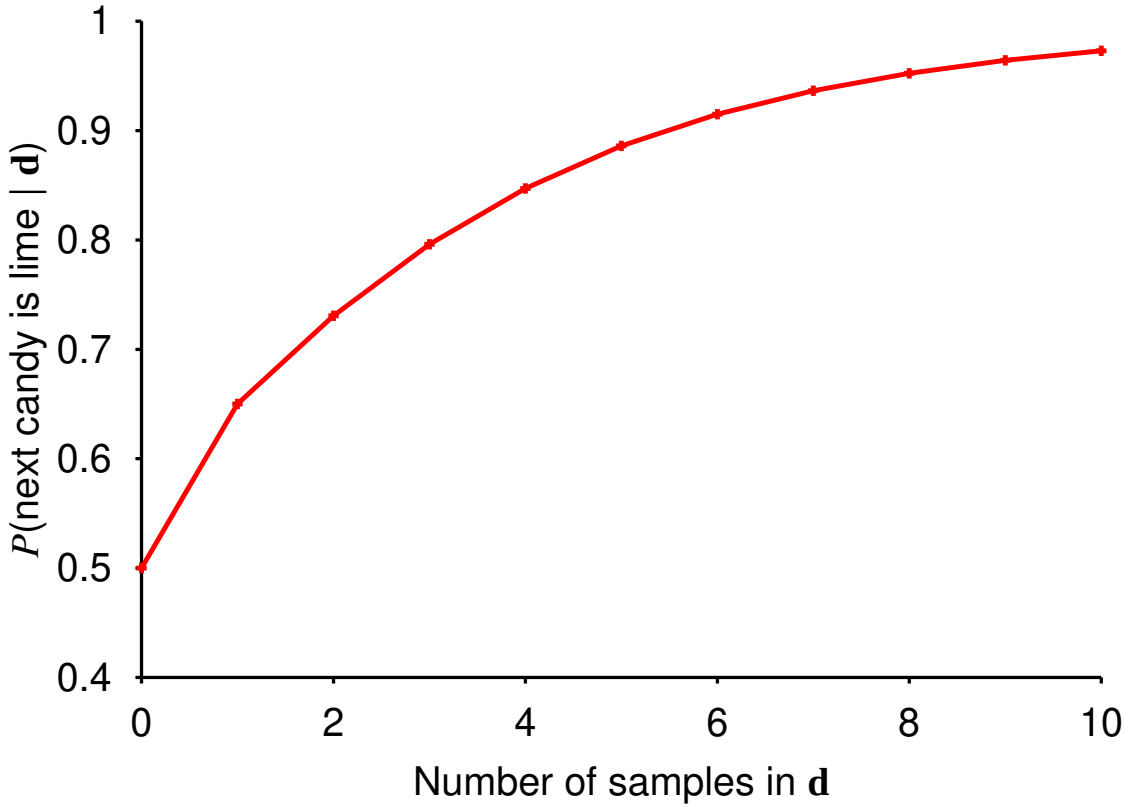
What kind of bag is it? What flavour will the next candy be?

$P(\mathbf{d}|h_i) = \prod_j P(d_j|h_i)$ (i.i.d. assumption)

Posterior probability of hypotheses



Prediction probability



- Agrees with the true hypothesis
- Optimal (given the hypothesis prior, another prediction is less right)

MAP approximation

Summing over the hypothesis space is often intractable
(Recall in DT, e.g., 18,446,744,073,709,551,616 Boolean functions of 6 attributes) $\Leftarrow$ limit of Bayesian learning $\Rightarrow$ approximation

Maximum a posteriori (MAP) learning: choose h_{MAP} maximizing $P(h_i|\mathbf{d})$

i.e., $\arg\max_i P(\mathbf{d}|h_i)P(h_i)$ or $\arg\max_i \log P(\mathbf{d}|h_i) + \log P(h_i)$

Log terms can be viewed as (negative of)

bits to encode data given hypothesis + bits to encode hypothesis

e.g., no bits are required such as h_5, $\log_2 1 = 0$

This is the basic idea of minimum description length (MDL) learning

ML approximation

For large datasets, prior becomes irrelevant

(no reason to prefer one hypothesis over another a priori)

Maximum likelihood (ML) learning (ML estimate): choose h_{ML} maximizing $P(\mathbf{d}|h_i)$

i.e., simply get the best fit for the data

identical to MAP for uniform prior ($P(\mathbf{d}|h_i)P(h_i) = P(\mathbf{d}|h_i)$)

ML is the non-Bayesian statistical learning

Parameter and structure

Parameter learning: finding the numerical parameters for a probability model whose structure is fixed

e.g., learning the conditional probabilities in a Bayesian network with a given structure

ML parameter learning: maximizing $L(\mathbf{d}|h_\theta) = \log P(\mathbf{d}|h_\theta)$, θ is a parameter

by $\frac{dL(\mathbf{d}|h_\theta)}{d\theta} = 0 \Leftarrow$ numerical optimization

Structure learning: finding the structure of a probabilistic model (e.g., Bayes net) from data by fitting the parameters

Bayesian structure learning: search for a good model by adding the links and fitting the parameters (e.g., hill-climbing etc.)

Bayes classifier#

PGM (Bayesian network) makes inference and learning tractable
for binary n variables, reduce the joint distribution from 2^n to $2n$

Naive Bayes classifier: features are conditionally independent of each other, given the class

$$P(h_i|\mathbf{d}) \propto P(\mathbf{d}|h_i)P(h_i) = \prod_j P(d_j|h_i)P(h_i)$$

- Learning: maximize likelihood by ML estimate — training
- Inference: predict the class by performing inference applying Bayes' rule — test

Can be applied to Bayesian networks (PGM)

Generative vs discriminative models

Two approaches to classification

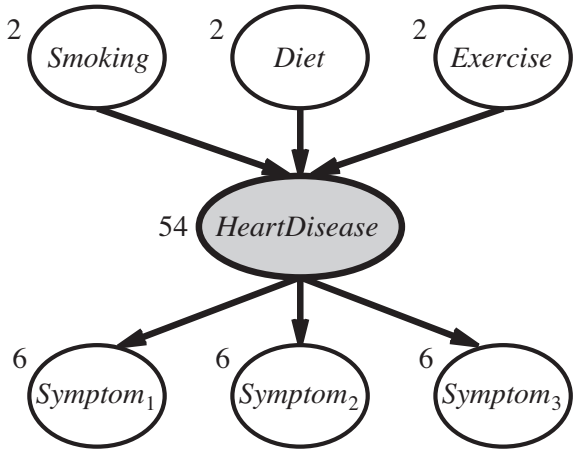
- Generative model: model the distribution of inputs given the target
  To solve: what does each class "look" like?
  - Build a model of $P(\mathbf{d}|h)$
  - Apply Bayes rule (say, Bayes classifier etc.)

- Discriminative model: estimate the conditional distribution (parameters) of the target given the input
  To solve: how do it separate the classes?
  - Learn $P(h|\mathbf{d})$ directly
  - From inputs (labeled examples) to classes (say, decision tree etc.)

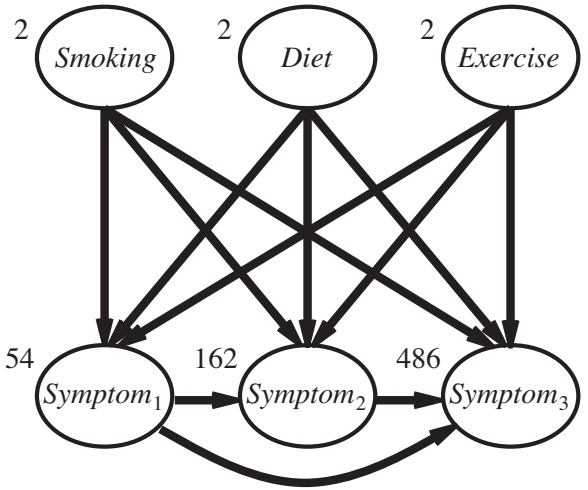
Expectation maximization*

EM (expectation-maximization): learning a probability model with hidden (latent) variables

- not observable data, causal knowledge
- dramatically reduce the parameters (Bayesian net)



(a)



(b)

(a) 78 parameters,

(b) 708 parameters

Clustering*

Unsupervised clustering: discerning multiple categories in a dataset

– unsupervised because the category labels are not given

– clustering by the data generated from a mixture distribution

$P(\mathbf{x}) = \sum_{i=1}^{k} P(C = i)P(\mathbf{x} \mid C = i)$

– – a distribution has k components (r.v. C), each of which is a distribution (say multivariate Gaussian, and so Gaussians mixture model (GMM))

– – $\mathbf{x}$ refers to the values of the attributes for a data point

– – – fit the parameters of a Gaussian by the data from a compnt.

– – – assign each data to a component by the parameters

Problems: we know neither the assignments nor the parameters

$\Leftarrow$ how to generate?

Gaussians mixture model*

Gaussians mixture model (GMM): most common mixture model

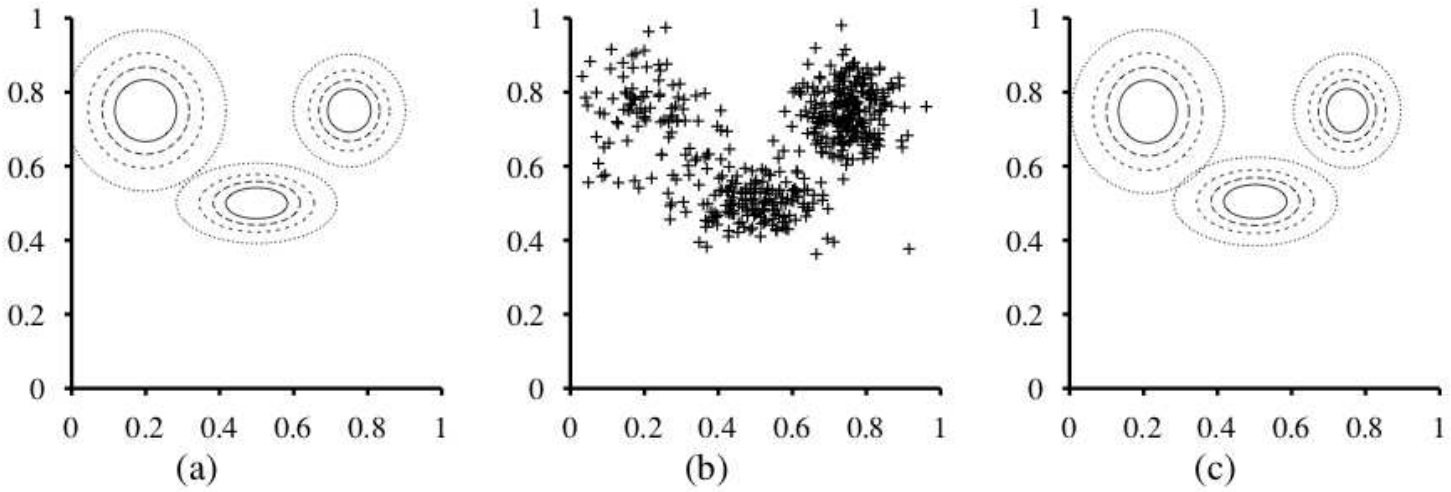
A GMM represents a distribution as

$$P(\mathbf{x}) = \sum_{i=1}^{k} \pi_i \mathcal{N}(\mathbf{x} \mid \mu_i, \Sigma_i)$$

with π_i the mixing coefficients, where $\sum_{i=1}^{k} \pi_i = 1$ and $\pi_i \geq 0$

- GMM is a density estimator
- **Theorem**: GMMs are universal approximators of densities (if there are enough Gaussians). Even diagonal GMMs are universal approximators
- In general mixture models are very powerful but harder to optimize $\Leftarrow$ EM

Example: EM*



(a) A Gaussian mixture model with three components
(b) 500 data points sampled from the model in (a)
(c) The model reconstructed by EM from the data in (b)

EM Algorithm*

Idea: pretend that we know the parameters of the model
infer the probability that each data belongs to each compnt.
refit the components to the entire data set
with each data weighted by the probability that it belongs to
the process iterates until convergence

1. E-step: Compute the posterior probability over C given current model $\Leftarrow$ deriving as expectation
2. M-step: Assuming that the data really was generated this way, change the parameters of each Gaussian to maximize the probability that it would generate the data it is currently responsible for $\Leftarrow$ maximum likelihood

EM Algorithm*

General form of EM algorithm

$$\theta^{(i+1)} = argmax_\theta \sum_{\mathbf{c}} P(\mathbf{C} = \mathbf{c} \mid \mathbf{x}, \theta^{(i)}) L(\mathbf{x}, \mathbf{C} = \mathbf{c} \mid \theta)$$

– θ: the parameters for the probability model
– $\mathbf{C}$: the hidden variables
– $\mathbf{x}$: the observed values in all the examples
– L: Bayesian networks, HMMs etc.

Derive closed form updates for all parameters

EM Algorithm*

1. Initialize the mixture-model parameters arbitrarily, for GMM
2. Iterate until convergence:

E-step: Compute the probabilities $p_{ij} = P(C = i \mid x_j)$, where the datum x_j was generated by component i

i.e., $p_{ij} = \alpha P(x_j \mid C = i)P(C = i)$ (Bayes' rule)

where $P(x_j \mid C = i)$ is the probability at x_j of the ith Gaussian

$w_i = P(C = i)$ is the weight for the ith Gaussian

define $n_i = \sum_j p_{ij}$, the number of data points assigned to i

M-step: Compute the new mean, covariance, and component weights using the following steps in sequence

$\mu_i \leftarrow \sum_j p_{ij}\mathbf{x}_j/n_i$

$\Sigma_i \leftarrow \sum_j p_{ij}(\mathbf{x}_j - \mu_i)(\mathbf{x}_j - \mu_i)^T/n_i$

$w_i \leftarrow n_i/N$, where N is the total number of data points

3. Evaluate log-likelihood and check for convergence

$lnP(\mathbf{x} \mid \pi, \mu, \Sigma) = \sum_{n=1}^{N}(\sum_{i=1}^{k} ln(\pi_i \mathcal{N}(\mathbf{x}_n \mid \mu_i, \Sigma_i))$

Transfer learning*

Assumption of learning already: the (training and test) data are drawn from the same feature space and distribution

may not hold in many real-world applications

Transfer learning (knowledge transfer): learning in one domain, but only having training data in another domain where the data may be in a different feature space or distribution

– greatly improve the performance of learning by avoiding expensive data labeling efforts

– people can intelligently apply knowledge learned previously to solve new problems (learning to Learn or meta-learning)

- What to transfer
- How to transfer
- When to transfer

Inductive transfer learning

ITL: the target task is different from the source task, no matter when the source and target domains are the same or not

– labeled data in the source domain are available (instance-transfer)

– – there are certain parts of the data that can still be reused together with a few labeled data in the target domain

– labeled data in the source domain are unavailable while unlabeled data in the source domain are available (self-taught learning)

Inductive instance transfer learning

Assume that the source and target domain data use the same set of features and labels, but the distributions of the data in the two domains are different (say, classification)

– some of the source domain data may be useful in learning for the target domain but some of them may not and could even be harmful

1. start with the weighted training set, each example has an associated weight (importance)
2. attempt to iteratively re-weight the source domain data to reduce the effect of the "bad" source data while encouraging the "good" source data to contribute more for the target domain
3. for each round of iteration, train the base classifier on the weighted source and target data, the error is only calculated on the target data
4. update the incorrectly classified examples in the target domain and the incorrectly classified source examples in the source domain

Ensemble learning*

An ensemble of predictors is a set of predictors whose individual decisions are combined in some way to classify new examples

E.g., (possibly weighted) majority vote

For this to be nontrivial, the classifiers must differ somehow, e.g.

Different algorithm

Different choice of hyperparameters

Trained on different data al Trained with a different weighting of the training examples

Ensembles are usually trivial to implement. The hard part is deciding what kind of ensemble you want, based on your goals

Bagging learning

Train classifiers independently on random subsets of the training data

Bagging (bootstrap aggregation)

Take a single dataset D with n examples

Generate m new datasets, each by sampling n training examples from D, with replacement

Average the predictions of models trained on each of these datasets

Random forests = bagged decision trees, with one extra trick to decorrelate the predictions

When choosing each node of the decision tree, choose a random set of d input features, and only consider splits on those features

Boosting learning

Train classifiers sequentially, each time focusing on training data points that were previously misclassified

Weak learner is a learning algorithm that outputs a hypothesis (e.g., a classifier) that performs slightly better than chance, e.g., it predicts the correct label with probability 0.6

not capable of making the training error very small

Can we combine a set of weak classifiers in order to make a better ensemble?

We are interested in weak learners that are computationally efficient

Decision trees

Even simpler: Decision Stump

– a decision tree with only a single split

AdaBoost algorithm

AdaBoost (Adaptive Boosting)

1. At each iteration we re-weight the training samples by assigning larger weights to samples (i.e., data points) that were classified incorrectly

2. We train a new weak classifier based on the re-weighted samples

3. We add this weak classifier to the ensemble of classifiers. This is our new classifier

4. We repeat the process many times

The weak learner needs to minimize weighted error

AdaBoost reduces bias by making each classifier focus on previous mistakes

Federated learning*

Federated learning (FL): many clients (edge device, e.g., mobile phones or IoT devices) collaboratively train a model under the orchestration of a central server while keeping the training data decentralized

– Ref: McMahan H et al., Communication-efficient learning of deep networks from decentralized data, arxiv, 2016

– Challenges: an unbalanced and non-iid data partitioning across a massive number of unreliable devices with limited communication bandwidth

FL Principles

– focused collection and data minimization

– systemic privacy risks

– lower costs than traditional centralized learning

E.g., Google Gboard mobile keyboard, TensorFlow Federated

Federated vs. peer-to-peer learning

FL vs. fully decentralized (peer-to-peer, p2p) learning

– Orchestration

– – a central orchestration server or service organizes the training, but never sees raw data vs. no centralized orchestration

– Wide-area communication

– – Hub-and-spoke (with the hub representing a coordinating service provider and the spokes connecting to clients) vs. peer-to-peer (with a possibly dynamic connectivity graph)

E.g., p2p in blockchain

Often applied to similar problems, say, a central authority may still be in charge of p2p learning

Data partitioning

Goal of FL: to learn a single global model that minimizes the empirical risk function over the entire training dataset (i.e., the union of the data across all the clients)

Using deep learning (by SGD), say, Federated Averaging algorithm (local-update or parallel SGD)

In FL, data can be non-iid in many ways

– dependence and non-identicalness: due to each client corresponding to a particular user/location/time

– if the data is in an insufficiently-random order, e.g. ordered by time, then independence is violated locally as well

E.g., client devices typically need to meet eligibility requirements in order to participate in training

The data is assumed to be partitioned by examples or relevant features

Federated averaging algorithm

def SERVEREXECUTE($client\ data$)
  local variables: M, clients per round
    T, total communication rounds
    initialize x_0 // all clients have the same amount of data
    for each round $t = 1, 2, \ldots, T$ **do**
    $S_t \leftarrow$ random set of M clients
    for each client **in** S_t **in parallel do**
      $x_{t+1}^i \leftarrow \text{CLIENTUPDATE}(i, x_t)$
      $x_{t+1} \leftarrow \sum_{k=1}^{M} \frac{1}{M} x_{t+1}^i$
    return *global data*
def CLIENTUPDATE(i, x)
  local variables: K, local steps per round
  for local step $j = 1, \ldots, K$ **do**
    $x \leftarrow x - \eta \nabla f(x; z)$ **for** $z \sim \mathcal{P}_i$ // local SGD
    return *local data*

Explanation-based learning*

Explanation-Based Learning (EBL) is a method of generalization that extracts general rules from individual observations (specific examples)

Idea: knowledge representation + learning

The knowledge-free inductive learning persisted for a long time (until the 1980s), — and NOW

Learning agents that already know something (background knowledge) and are trying to learn some more (incremental knowledge)

Formalizing learning

$Descriptions$: the conjunction of all the examples in the training set
$Classifications$: the conjunction of all the example classifications
$Hypothesis \wedge Descriptions \models Classification$

$Hypothesis$ the explains the observation must satisfy the entailment constraint

$Hypothesis \wedge Descriptions \models Classification$
$Backgroud \models Hypothesis$

EBL: the generalization follows logically from the background knowledge

extracting general rules from individual observations

Note: It is a **deductive** form of learning and cannot by itself account for the creation of new knowledge

Formalizing learning

$Hypothesis \wedge Descriptions \models Classification$
$Backgroud \wedge Descriptions \wedge Classifications \models Hypothesis$

RBL (relevance-based learning): the knowledge together with the observations allows the agent to infer a new general rule that explains the observations $\Leftarrow$ reduce version spaces

$Background \wedge Hypothesis \wedge Descriptions \models Classification$

KBIL (knowledge-based inductive learning): the background knowledge and the new hypothesis combine to explain the examples

also known as inductive logic programming(ILP)

representing hypotheses as logic programs

E.g., a Prolog-based speech-to-speech translation (between Swedish and English) was real-time performance only by EBL (parsing process)

Formalizing learning

Hypotheses

Hypothesis space $H = \{H_1, \cdots, H_n\}$ in which one of the hypotheses are correct

i.e., the learning algorithm believes $H_1 \vee \cdots \vee H_n$

Formalizing learning

Examples

$Q(X_i)$ if the example is positive

$\neg Q(X_i)$ if the example is negative

Extension: each hypothesis predicts that a certain set of examples will be examples of the goal (predicate)

– two hypotheses with different extensions are inconsistent with each other

– as the examples arrive, hypotheses that are inconsistent with the examples can be ruled out

Version spaces algorithm

```
def VERSION-SPACE-LEARNING(examples)
    local variables: V, the version space: the set of all hypotheses

    V ← the set of all hypotheses
    for each example e in examples do
        if V is not empty then V ← VERSION-SPACE-UPDATE(V, e)
    end
    return V // a version space
def VERSION-SPACE-UPDATE(V, e)
    V ← {h ∈ V : h is consistent with e}
    return V
```

Find a subset of V that is consistent with all the *examples*

EBL

1. Given an example, construct a proof that the goal predicate applies to the example using the available background knowledge

– "explanation": logical proof, any reasoning or problem-solving process

2. In parallel, construct a generalized proof tree for the variability goal using the same inference steps as in the original proof

3. Construct a new rule whose left-hand side consists of the leaves of the proof tree and whose right-hand side is the variability goal

4. Drop any condition from the left-hand side that is regardless of the values of the variables in the goal

Need to consider the efficiency of EBL process

Computational learning theory*

How do we know that h is close to f if we don't know what f is??

How many examples do we need to get a good h??

How complex should h be??

Computational learning theory analysis the sample complexity and the computational complexity of (inductive) learning

There is a trade-off between the expressiveness of the hypothesis language and the ease of learning

Probably approximately correct*

Principle: any hypothesis that is consistent with a sufficiently large set of examples is unlikely to be seriously wrong

Probably Approximately Correct (PAC)

– h is approximately correct if $error(h) \leq \varepsilon$ (a small constant)

PAC learning algorithm: any learning algorithm that returns hypotheses that are probably approximately correct

– aims at providing bounds on the performance of various learning algorithms

The Curse of dimensionality*

Low-dimensional visualizations are misleading

In high dimensions, "most" points are far apart

KNN: In high dimensions, "most" points are approximately the same distance

No free lunch*

Theorem (Wolpert, 1996): averaged over all possible data-generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points

$\Rightarrow$ no learning algorithm is universally any better than any other

The goal is not to seek a universal learning algorithm or the absolute best learning algorithm

Instead, the goal is to understand what kinds of distributions are relevant to the "real world"

Universal approximations*

Recall

Theorem: All continuous functions w/2 layers, all functions w/3 layers

Theorem: GMMs are universal approximators of densities (if there are enough Gaussians). Even diagonal GMMs are universal approximators

Theorem: Any function computable by a Turing machine can be computed by an RNN of a finite size

How about that??

Are those learning real learning??

Learnability*

Learnability can be undecidable (ref. Ben-David et al. Nature, 2019)

Theorem: The EMX (estimating the maximum) learnability of F^* w.r.t. P^* is independent of the ZFC axioms

– The family of sets F^* is the family of all finite subsets of the interval $[0, 1]$

– The class of probability distributions P^* is the class of all distributions over $[0, 1]$ with finite support (measurability)

Discovery*

ML can aid mathematicians in discovering new conjectures and theorems in pure math

- discovering and proving one of the first relationships between algebraic and geometric invariants in knot theory (Low-dimensional topology)

- conjecturing a resolution to the combinatorial invariance conjecture for symmetric groups (representation theory)

Algorithm

- deep learning: a fully connected feed-forward neural network, with hidden unit sizes [300, 300, 300] and sigmoid activations - the task: a multi-class classification problem

Ref. Davies A. et al., Advancing mathematics by guiding human intuition with AI. Nature 600, 70–74 (2021)

Smale's 18th problem*

What are the limits of intelligence, both artificial and human??

Theorem There are well-conditioned problems where accurate neural networks exist, but no algorithm can compute them

Colbrook M. et al., The difficulty of computing stable and accurate neural networks: On the barriers of deep learning and Smale's 18th problem, PNAS, 2022