Chapter 18: Learning

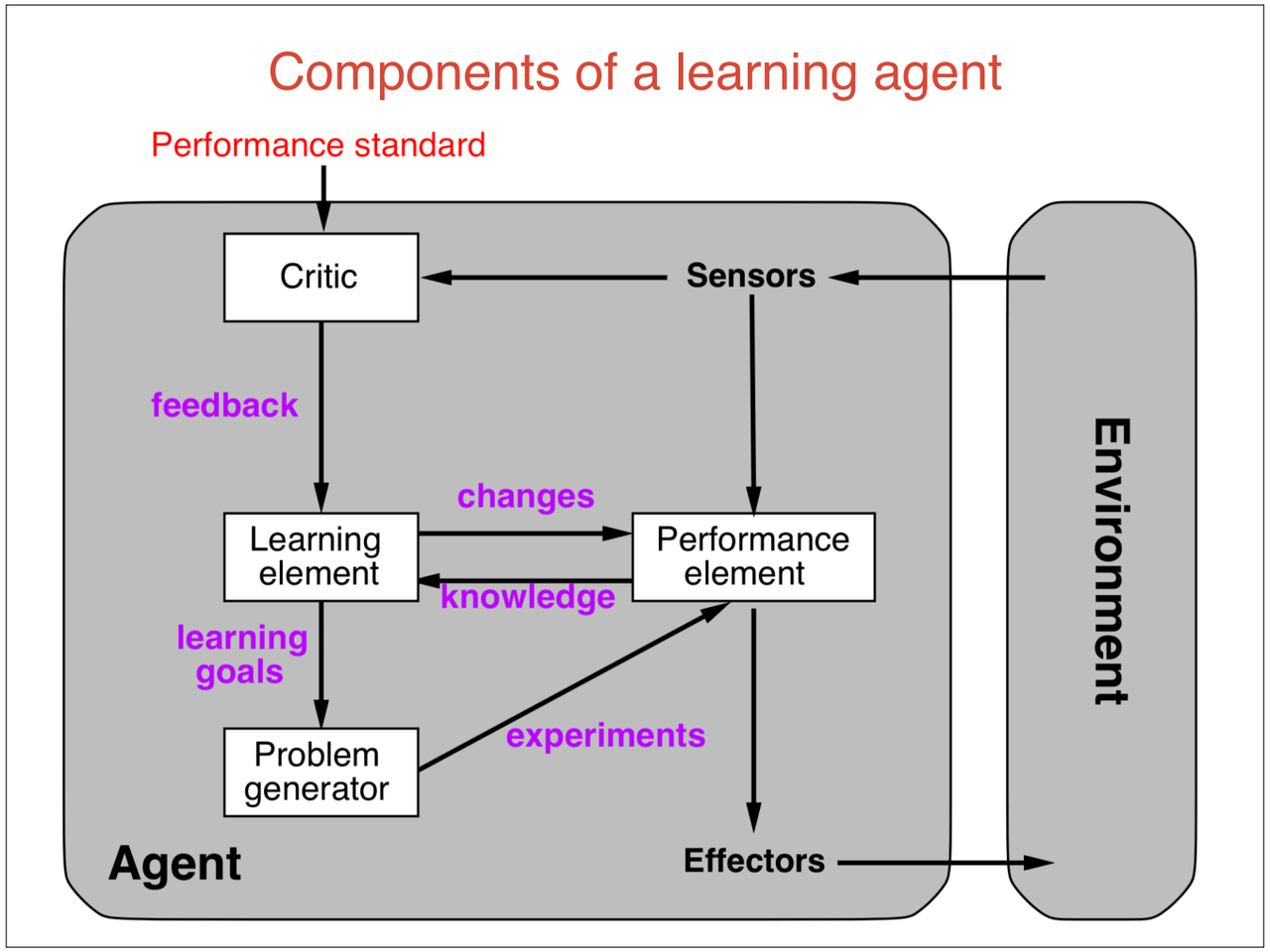
W46:
- No lecture Mon, guest lecture Wed
- Debugging tips for AI code:
- Code carefully and reread.
- Work through an example by hand.
- Come up with examples where you can figure out the solution.

Machine learning: Methods that improve with experience.
So far we have designed a fixed algorithm and used that.
Often it's too hard to come up with a good program ourselves; it works better to

We will focus on simplest case:
- Episodic = each action is separate.
- Also discrete, deterministic

Example: filtering spam email.

Machine learning is everywhere now:
- advertising
- recommendations
- interface choices
- vision, image processing
- voice, text processing
- game playing, robots



Performance element: The actual program that chooses actions.
Critic+learning element: A program that changes the performance element based on feedback.

Design of a learning agent

Feedback: How do we get feedback on whether choices were good or not?

Choices:
- What type of performance element is used (i.e. what algorithm).
- What component of the performance element is learnable.
- What representation of that component do we manipulate.

Google map search. Feedback: Time to destination.
- Performance element: BFS. Component to learn: None!
- Performance element: A*. Component to learn: heuristic h(). Representation: which distance metric.

Chess: Feedback: Win/loss.
- Performance element: minimax. Component: evaluation function. Representation: weighted linear function (i.e. value of each piece).

Spam filter: Feedback: User click "mark as spam"
- Performance element: blacklist filter. Component: blacklist. Representation: words on blacklist.
- Performance element: weighted linear function (logistic regression). Component: function. Representation:

Inductive learning

Inductive learning (AKA science): How we learn from experience.

Simplest example: there is some function f(x).
- i.e.: f(spam email) = spam / not spam. f(chess board) = optimal move.

We want a hypothesis h(x) ~= f(x)
We have a training set f(x^(i)) = y^(i)

Example: curve fitting with a polynomial.

Central problem in learning:
- Expressiveness vs generalizability of the model.
- AKA bias vs variance. AKA training error vs test error.
- A more expressive model (i.e. more polynomial terms) will be more consistent with the training data. Will it generalize to other data?
- Ockham's razor: maximize a combination of consistency with training data and simplicity.
- We'll talk more about evaluation later.

Feature-based representation

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*

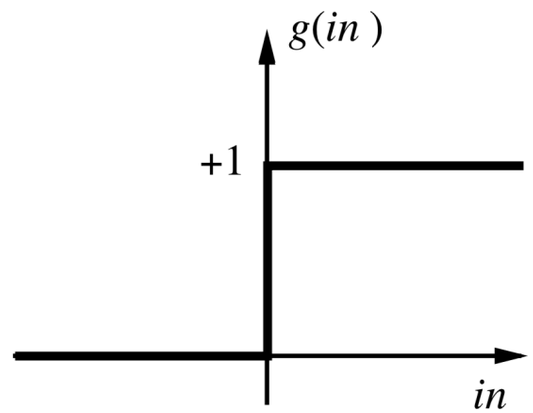
Problem: Predict whether I will decide to stay and wait for a table at a restaurant, or go find another one.

Feature-based representation: Each example is represented by a vector of features / attributes.

- Other cases: Spam email: list of words in the email
- Chess game: (1) list of living pieces on each side; (2) list of which piece is in which square.

Linear classifier (perceptron)

$$h(x_{1:J}) = g\left(\sum_j w_j x_j - w_0\right)$$

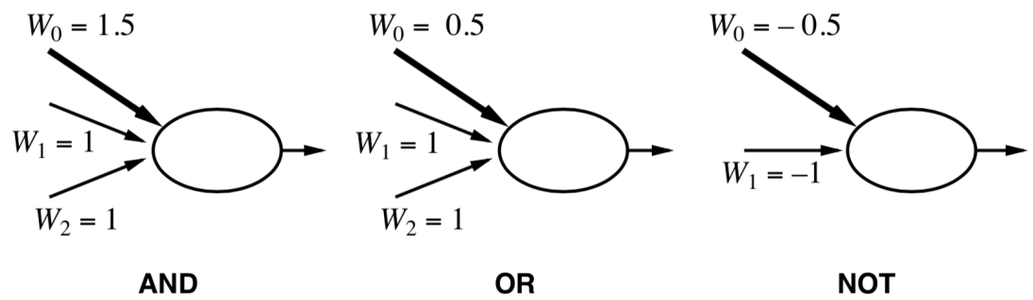


g: threshold function.

"in": input to the threshold function. We will use the index in_i later.

We learn by changing the weights w_{1:j}

Implementing logical functions



Gradient descent

Gradient descent is a local search algorithm.

Idea:

- Start with arbitrary weights w_{1:J}.
- At each iteration, move the weights in the direction that improves the error on the training set.
- Which direction to go? Take the derivative of the error with respect to each weight.

Perceptron learning

Learn by adjusting weights to reduce error on training set

The squared error 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 search by gradient descent:

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

Simple weight update rule:

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

We will use gradient descent.
Specifically, stochastic gradient descent: we will

Squared error:
- The most natural error is just 1 if we're wrong and 0 if we're right on a given example. But that is not differentiable, so if we can't use gradient descent.
- There are a few ways to "smooth" the error -- we're using squared error, which is probably the simplest.

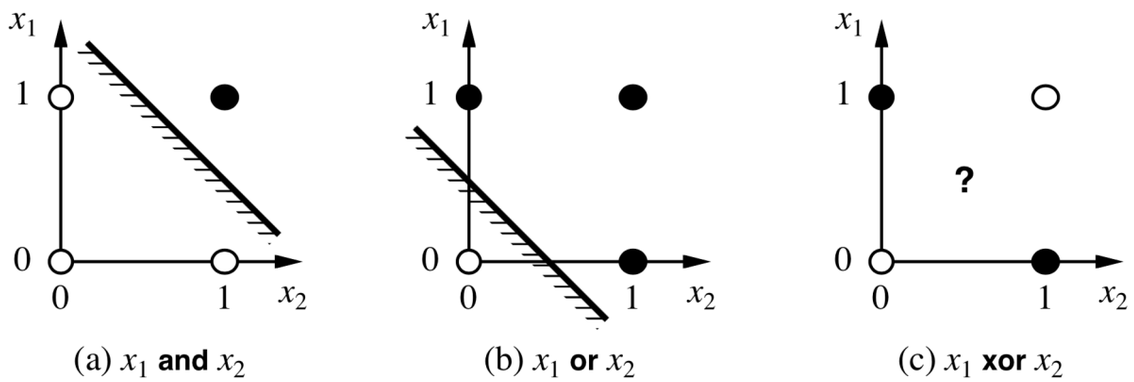
Derivative of the error: dE/dw_j = if I increase w_j, does that increase the error, and by how much?

Update formula: Note the sign. We're trying to decrease the error, so we subtract the derivative, so the minus sign cancels out.

A few considerations. There is no right way to make these choices:
- Initial weights.
- Learning rate alpha. Too small -> training will take a long time. Too large -> training won't work (it will diverge).

Perceptron learning algorithm:
- while not converged:
    - pick a training example
    - compute the derivative of the error on that example with respect to each weight.
    - update each weight using the formula

Expressiveness of perceptron

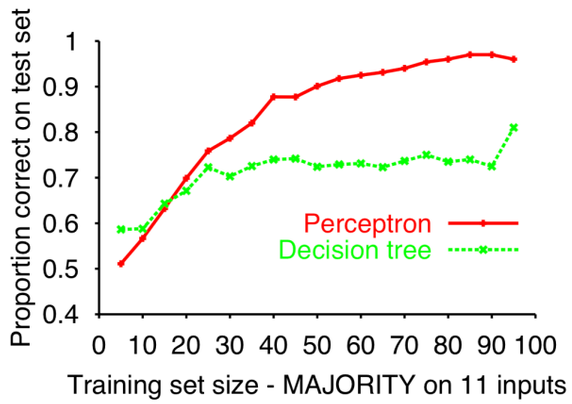


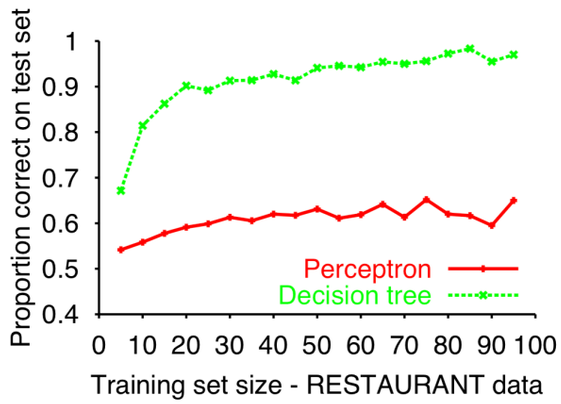
We can implement AND or OR
AND: w0 = 1.5; w1 = 1, w2 = 1
OR: w1 = 0.5; w1 = 1, w2 = 1

What about XOR?

Expressiveness of perceptron

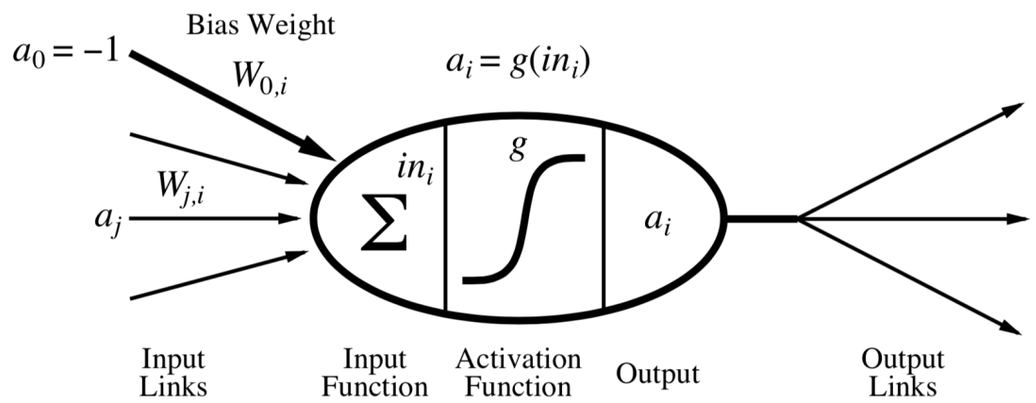
Perceptron learning rule converges to a consistent function
for any linearly separable data set





McCulloch–Pitts unit (AKA artificial neuron)

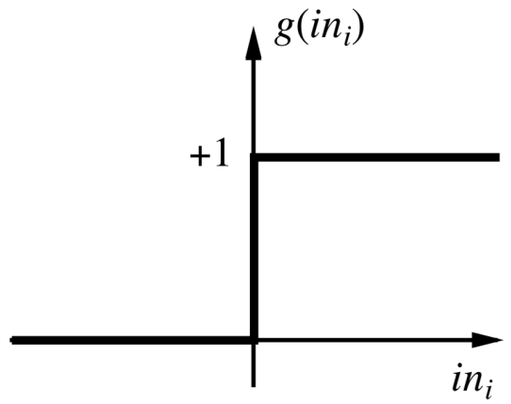
$$a_i \leftarrow g(in_i) = g\left(\Sigma_j W_{j,i} a_j\right)$$



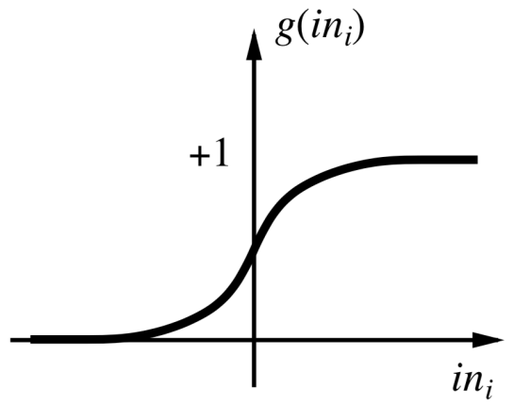
Inspired by the brain, but the similarity is weak.

Neural network unit (AKA artificial neuron, AKA node)

Activation functions



Threshold function

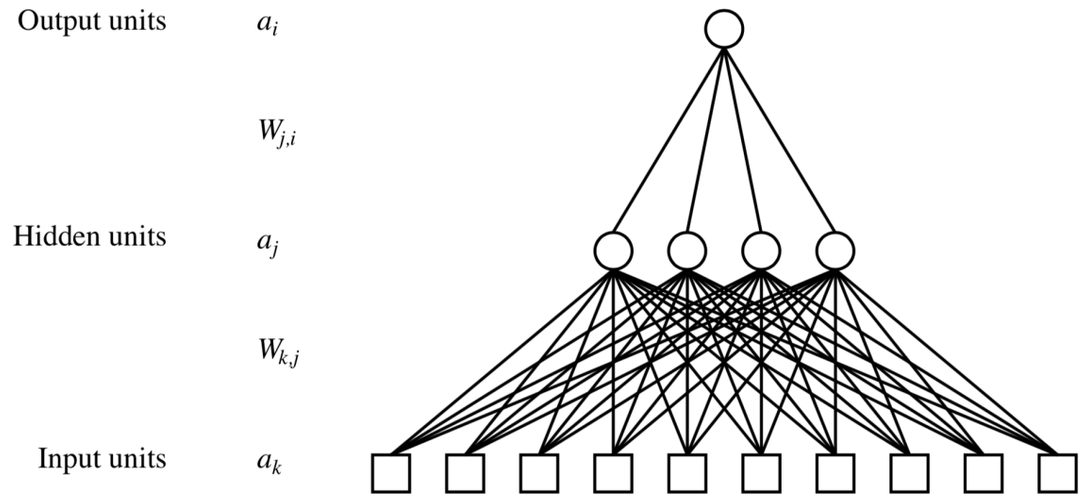


Logistic (AKA sigmoid) function

$$g(in_i) = 1/(1 + \exp(-in_i))$$

There are other activation functions that are used in practice. There is no right choice of function.

Multi-layer perceptron (MLP)



This is one architecture of neural network. We'll talk about other architectures later.

The number of hidden units is usually chosen by hand.

We can use more than one hidden layer.

Learning a MLP using backpropagation

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: **back-propagate** the error from the output layer:

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

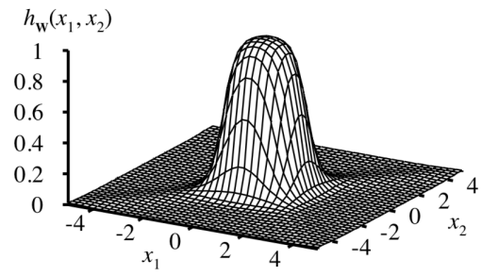
Update rule for weights in hidden layer:

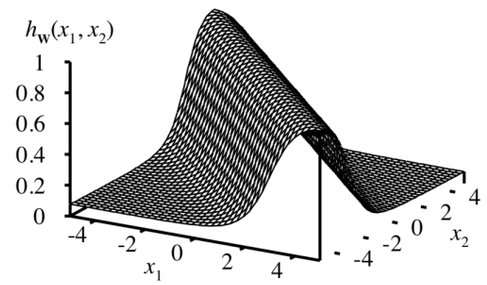
$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \ .$$

How can we compute the derivative of the weights of a MLP?

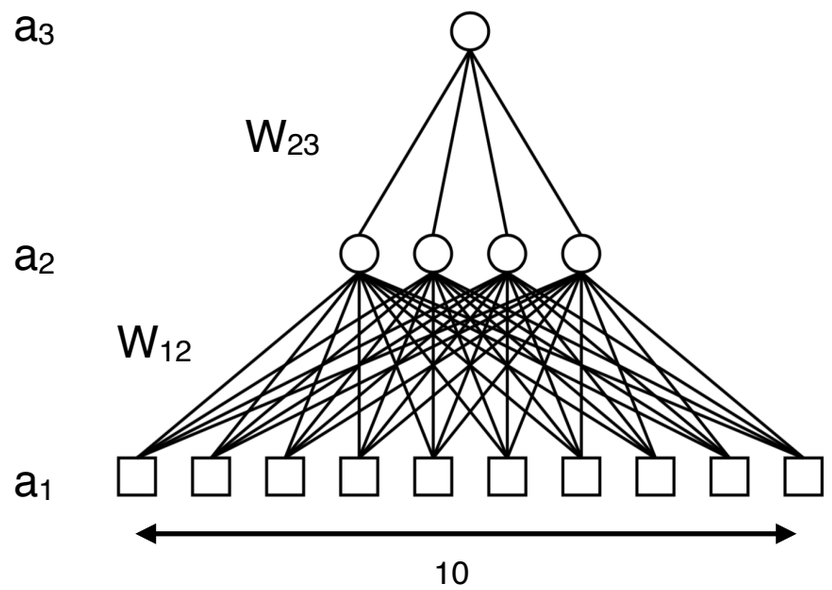
Backpropagation probably doesn't happen in the brain.

Any function can be expressed as an MLP with three layers



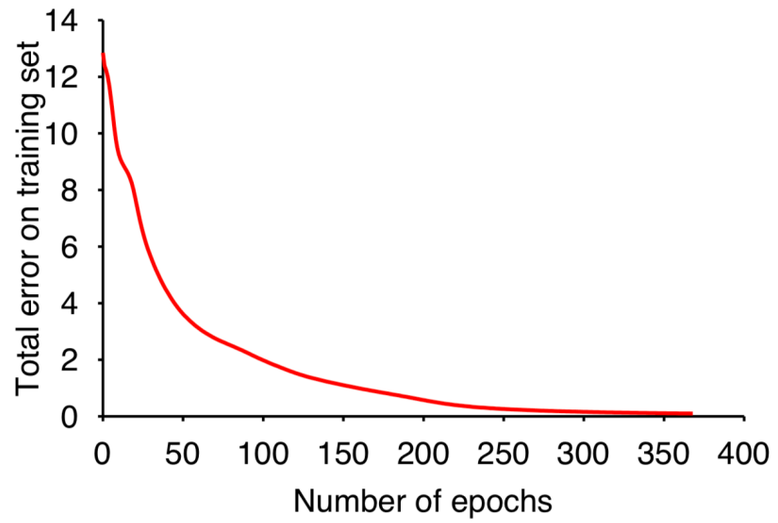


Matrix formulation



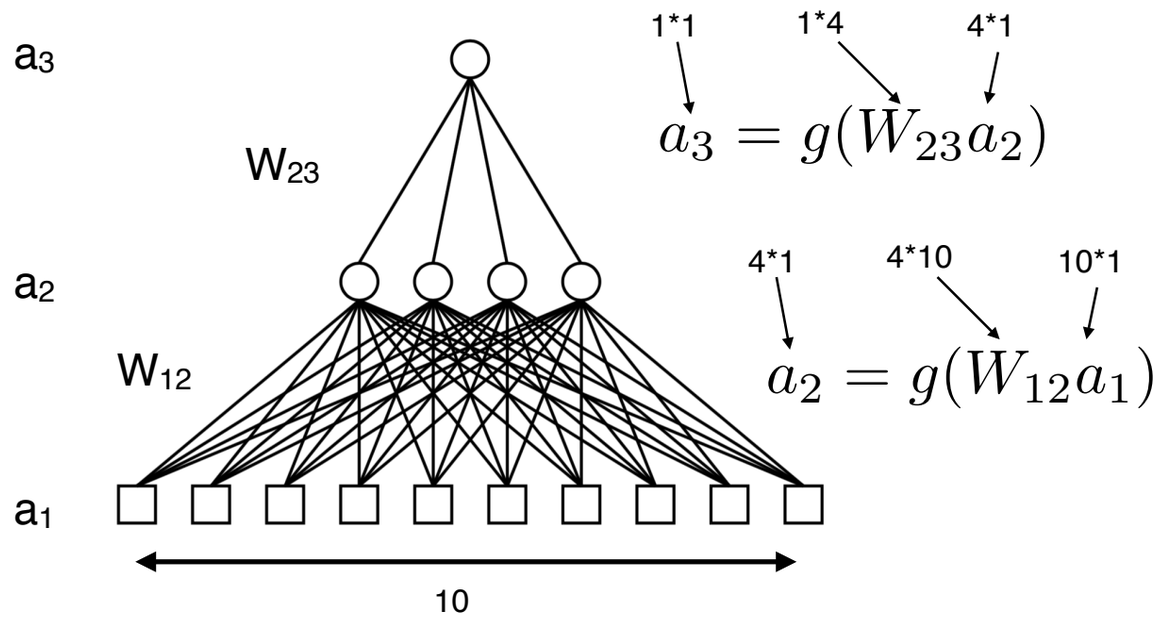
MLP learning

Training curve for 100 restaurant examples: finds exact fit



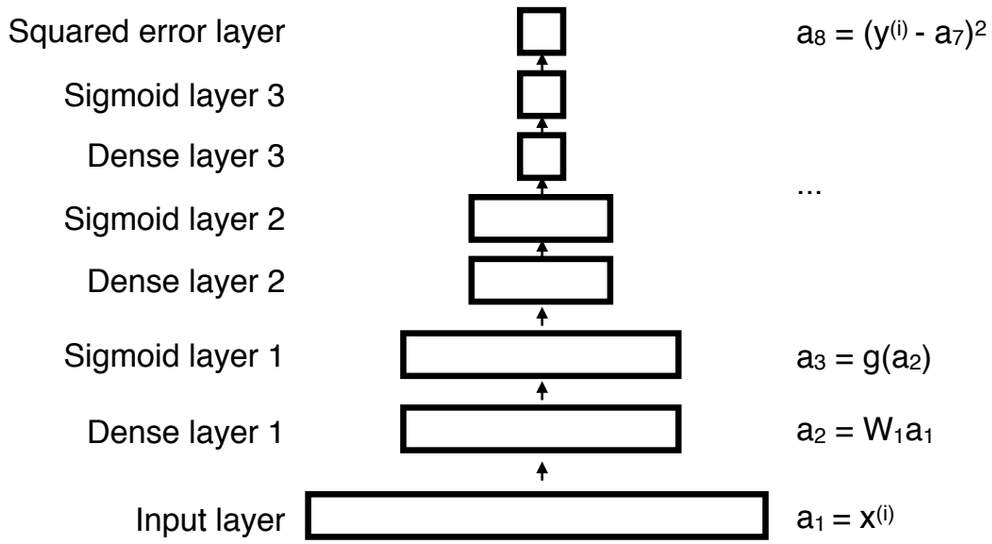
epoch: one iteration of stochastic gradient descent on each training example.

Matrix formulation

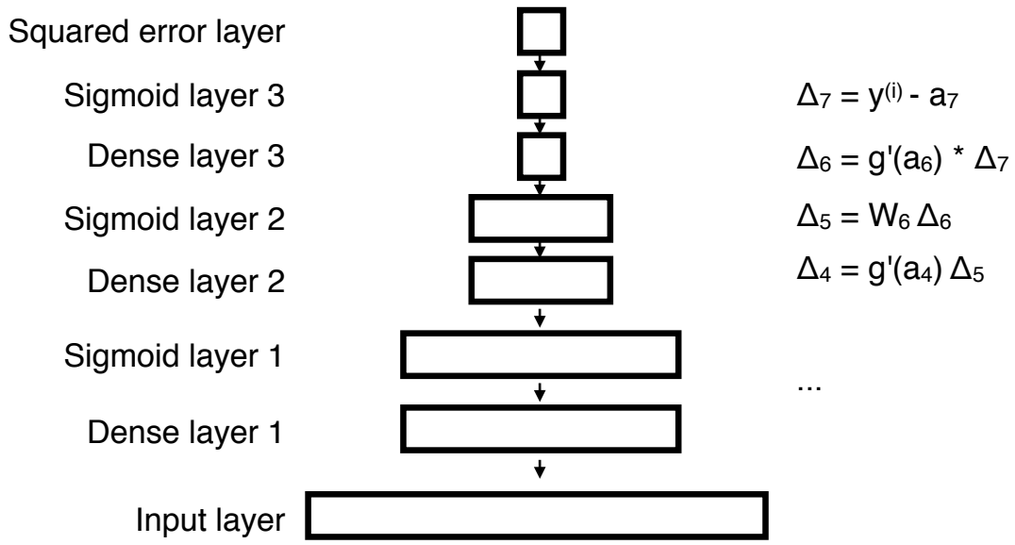


Matrix operations are much faster than loops!

Layers

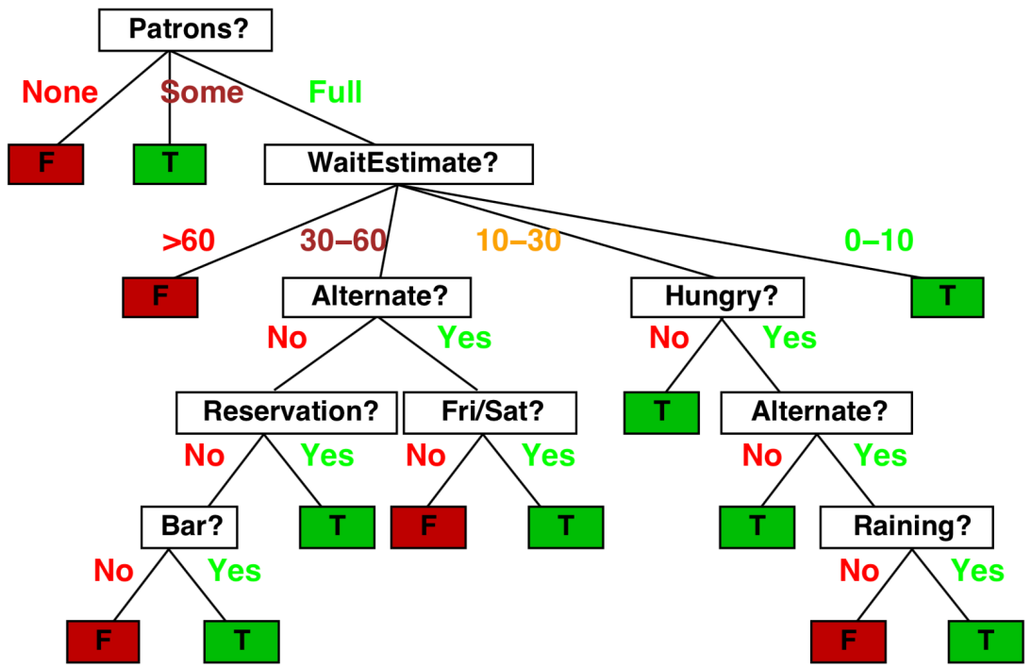

Squared error layer
Sigmoid layer 3
Dense layer 3
Sigmoid layer 2
Dense layer 2
Sigmoid layer 1
Dense layer 1
Input layer
$a_8 = (y^{(i)} - a_7)^2$
...
$a_3 = g(a_2)$
$a_2 = W_1 a_1$
$a_1 = x^{(i)}$

Backpropagation



No learning to do on sigmoid layers, sqerr layers.

Decision tree



Decision tree: One possible hypothesis h()

Expressiveness of decision trees

Expressiveness:
Decision trees can represent any function of the input.
-> One leaf for every possible set of inputs
- We can find a tree that is consistent with any training set (unless f(x) is nondeterministic)

How many possible Boolean functions (i.e. truth tables) on N Boolean features?
2^N possible assignments of inputs. Either T/F for each one
-> 2^2^N possible functions. 6 features -> ~10^19 possible functions!

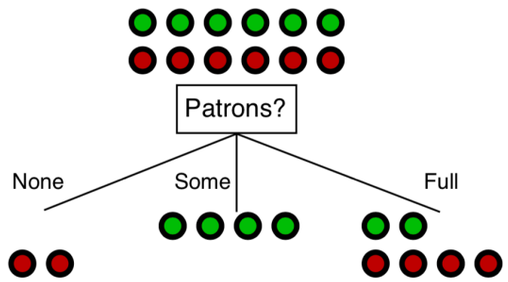
- Complexity vs generalizability: We want a compact decision tree -- e.g. limit maximum depth.

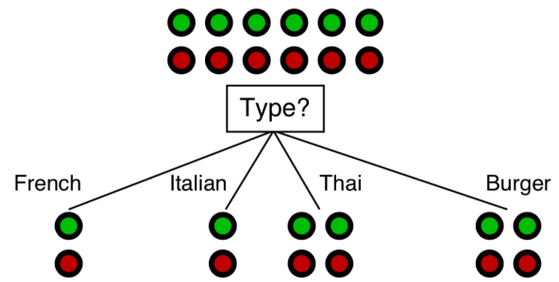
Decision tree learning

```
def decision_tree_learning(examples,
                           attributes=examples.attributes):
   if examples is empty: return default
   if all examples have the same label: return that label
   if attributes is empty: return mode(examples)
   best = choose_attribute(examples, attributes)
   tree = decision tree with root test best
   for each value v of best:
       examples = subset of examples with best=v
       subtree = decision_tree_learning(examples,
                                        attributes-best)
       tree = add subtree as a branch to tree with label v
   return tree
```

Choosing an attribute

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





Information

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

Entropy: a measure of how much uncertainty there is in a prediction.
Entropy is related to how many bits it takes to encode a message.
Two classes: H(p/(p+n) , n/(p+n))

H(p, 1-p) vs p: Zero for p = 0 or 1; one for p=0.5.

Choosing an attribute

$$\text{Expected entropy} = \sum_{\text{Leaf } i} N_i H(P_i, 1 - P_i)$$

$$P_i = \frac{T_i}{T_i + F_i}$$

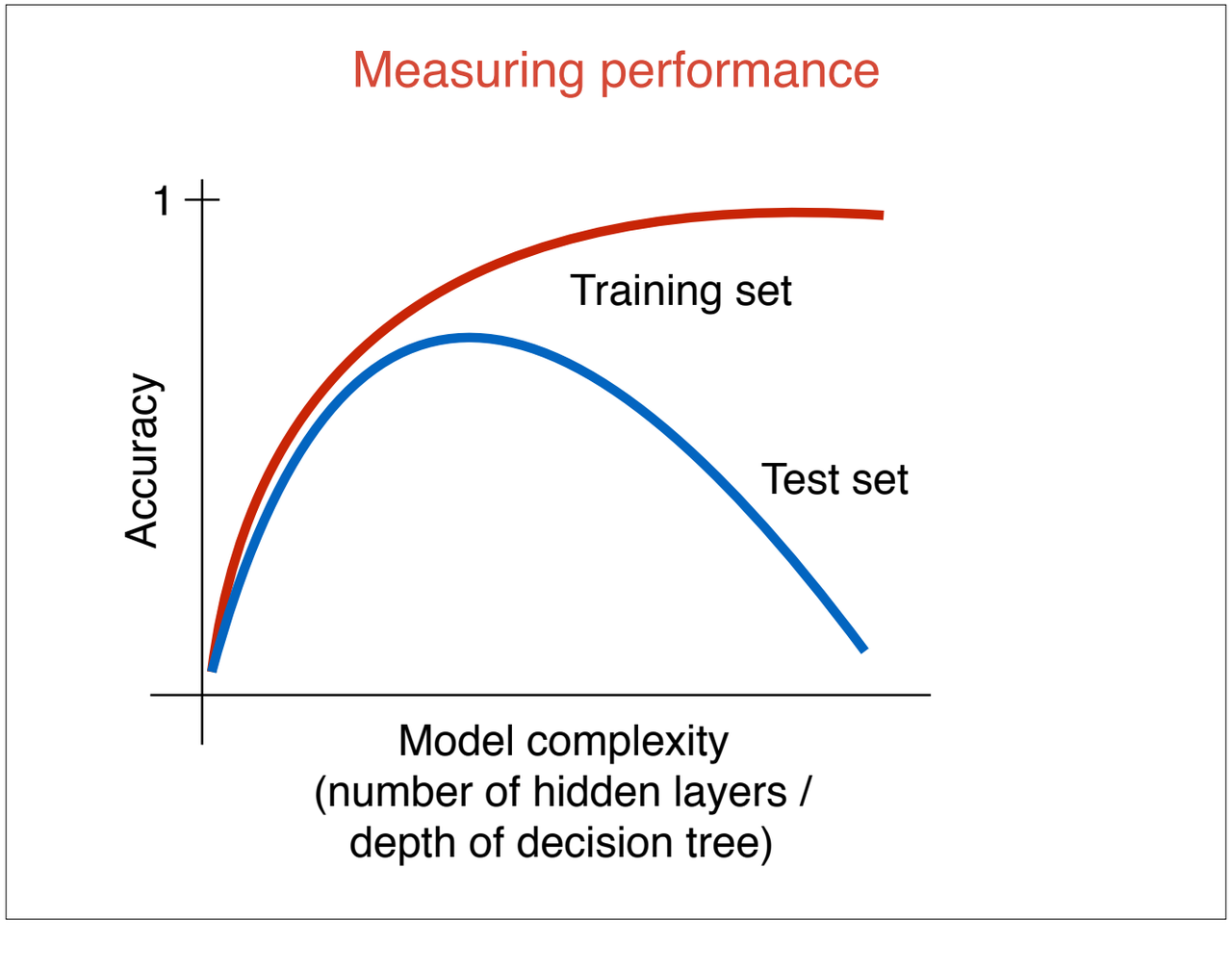
We choose the attribute with the lowest average entropy of children.

N_i: Number of examples at leaf i = T_i + F_i
P_i: Probability of true at leaf i
T_i: Number of true examples at leaf i
F_i: False examples

Measuring performance

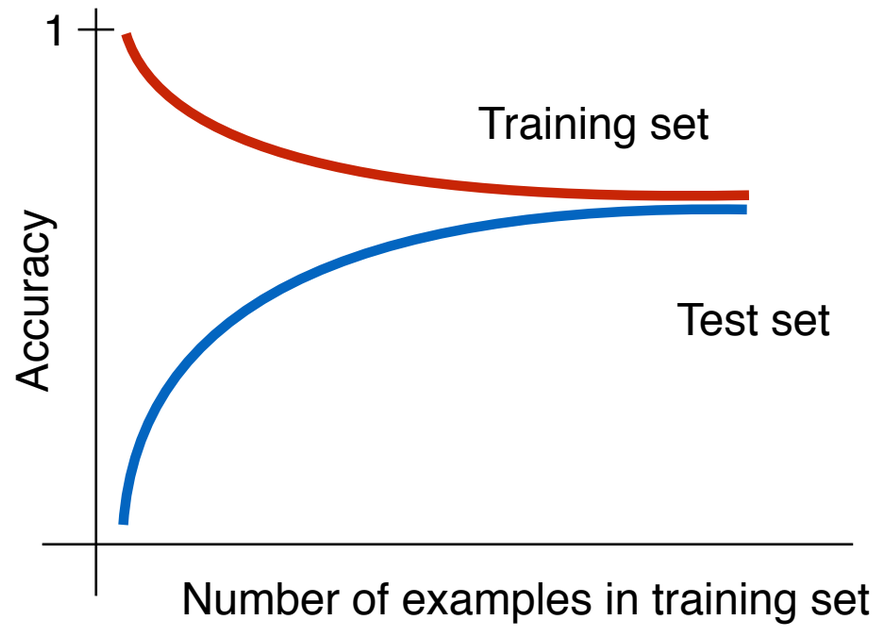
Performance measurement

- How to decide if h ~= f?
- We want to know if it works in the real world, not just on our training set.
- Two options: (1) use learning theory; (2) use accuracy on a test set.
- Train/test split.
    - Can do multiple splits. Cross-validation. Leave-one-out.



- Tradeoff: goodness of fit vs model complexity
- Accuracy ~ model complexity. i.e. decision tree depth / minimum number of examples per leaf. Fixed training set size.
--- Training set: Always increases.
--- Test set: (1) curve with peak. (2) worse model; peaks lower. Under/over-trained.

Measuring performance



- Learning curve: accuracy ~ training set size.
--- Training set: starts high, reduces.
--- Test set: Three options: slow convergence, quick convergence, convergence to non-perfect accuracy.

Once you use some data to test multiple models and pick the best one, it no longer will give you an accurate measure of your accuracy. Need three sets: training, validation, test.

- Performance measurement in practice:
--- Make sure evaluation set is a good model of real data. Example: lab biases in bacterial antibiotic resistance prediction.
--- What performance measurement is appropriate for your setting?

Question 1

Design a neural network that implements the XOR function. That is, define neurons, their connectivity and weights. Use threshold as your activation function (that is, the function that returns 0 or 1 if the input is negative or positive respectively).

Extra challenge: do so using the fewest number of hidden neurons.

Question 2

Imagine you have defined a neural network using the identity activation function; that is, g(x) = x. Suppose the network has one hidden layer with 3 neurons, which is fully connected to the input. Write the expression for a_out as a function of the input. Show that you can express the same network as a single-layer perceptron. What does this tell you about the expressive power of a multi-layer neural network with the identity activation function?

a_2 = W_12 a_1
a_out = W_23 a_2 = W_23 * W_12 * a_1

Equivalently, we could remove the hidden layer and use a perceptron with weights (W_23 * W_12)