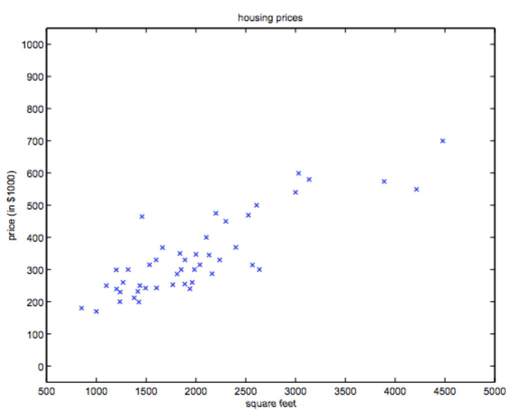
5. Supervised Learning

- Classification
- Regression

Supervised Learning

- suppose we have a dataset of living areas and prices of houses:

living area (ft^2)	price (1k\$)
2104	400
1600	330
2400	369
1416	232
⋮	⋮



- Supervised learning:
  - given n samples of paired data: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(n)}, y^{(n)})$
  - how can we predict the price of a new house with size x ?

- $x^{(i)} \in \mathbb{R}^{d_x}$ is called the "input" variable or input **features**
- $y^{(i)} \in \mathbb{R}^{d_y}$ is called the "output" variable or **target** variable
- the goal is to predict the target variable from input features
- **training set** $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$ is the set of data we have to be used to make the prediction
- note that the domain of x and y need not be real-valued vector spaces
- formally, our goal is to learn a function $h : \mathbb{R}^{d_x} \to \mathbb{R}^{d_y}$ such that $h(x)$ is a good predictor for y

- then, what is **unsupervised learning**?

- two types of supervised learning
  - **regression**
  - **classification**

Linear regression

- consider a slightly richer features

living area (ft^2)	# bedrooms	price (1k$)
2104	4	400
1600	3	330
2400	3	369
1416	2	232
⋮	⋮	⋮

- $x \in \mathbb{R}^2$ ($d_x = 2$), x_1 is the area, x_2 is the number of rooms
- depending on the problem we need to search for h from different classes of functions, but for now we focus on linear functions of the form

$$h_\theta(x) \;=\; \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

- θ_i's are the **parameters** (also called **weights**) parameterizing the space of linear functions from $\mathbb{R}^{d_x} \to \mathbb{R}$
- in general

$$h_\theta(x) \;=\; \sum_{i=0}^{d_x} \theta_i x_i \;=\; \theta^T x \;=\; \langle \theta, x \rangle$$

- in this class of linear functions, we need to define a measure for which function is better so that we can find the ebst one
- this measure of choice is called **cost function**, for example

$$J(\theta) \;=\; \sum_{i=1}^{n} \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

  which is defined over the given training data
- this choice of cost function is called an **ordinary least squares**, which can be solved very efficiently
- we are searching for a predictor h that best explains the given training data, and in general we will choose a **loss** $\ell : \mathbb{R} \to \mathbb{R}$ and define the cost as

$$J(\theta) \;=\; \sum_{i=1}^{n} \ell(h_\theta(x^{(i)}), y^{(i)})$$

- learning a predictor is now reduced to a functional estimation problem of finding the parameter θ that minimizes the cost $J(\theta)$
- we consider the **gradient descent** approach, which starts with some initial θ and repeatedly updates θ as per

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

- the step size α is called the **learning rate**
- this is a popular algorithm for taking the steepest descent direction for minimizing a function, and in a vector notation

$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$

- this algorithm can be applied as long as we can evaluate the gradient

- to implement the algorithm we need to evaluate the gradient of the cost function

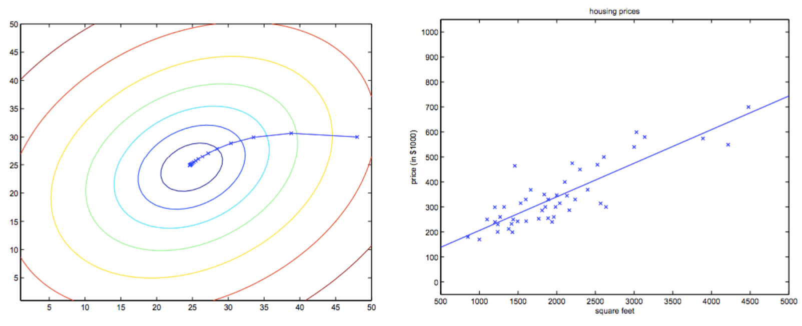
$$
\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \sum_{i=1}^{n} \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_\theta(x^{(i)}) - y^{(i)})^2 \\
&= \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)}) \times \frac{\partial}{\partial \theta_j} (h_\theta(x^{(i)}) - y^{(i)}) \\
&= \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)}) \times \frac{\partial}{\partial \theta_j} (\sum_{j=0}^{d_x} \theta_j x_j^{(i)} - y^{(i)}) \\
&= \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}
\end{aligned}
$$

- this gives the algorithm
  - repeat until convergence
  - for all $j \in \{0, \ldots, d_x\}$

$$\theta_j \leftarrow \theta_j - \alpha \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

> If the target $y^{(i)} = 1$ and the current estimate $h_\theta(x^{(i)}) = 0.5$ and $x_j^{(i)} = 2$, then we would want to increase θ_j such that $h_\theta(x^{(i)})$ increases. Gradient descent turns this intuition into a concrete learning algorithm.

- interpretation:
  - jointly for all coordinates but for each sample i, the update magnitude is proportional to the error $|h_\theta(x^{(i)}) - y(i)|$
  - if there is a training example with large error, we tend to fit it aggressively, but if there is a training example with small error then we do not change the parameter too much
- typical gradient descent gets stuck at local minima without further assumptions (such as convexity), but for our choice of $J(\theta)$ there is only one local minima which is the global minima



- what could go wrong with gradient descent?
- what could go wrong with quadratic cost $J(\theta)$?

- versions of gradient descent methods
  - **batch gradient descent** uses all the training examples to compute the gradient at every iteration
  - **stochastic gradient descent** uses a sub-samlped training examples of small cardinality
- **stochastic gradient descent**
  - repeat until convergence
  - for $i = 1$ to n
  - for all $j \in \{0, \ldots, d_x\}$

$$\theta_j \leftarrow \theta_j - \alpha(h_\theta(x^{(i)}) - y(i))x_j^{(i)}$$

- for large-scale real-world problems, stochastic gradient descent is preferred, mainly due to the empirical success in finding a better predictor faster
- theoretically understanding the power of stochastic descent is an important problem, and active topic for research

- note that this linear regression was an exercise to learn supervised learning in general
- for solving quadratic cost linear regression problems, there is a much simpler way
- note that the data can be represented as (we let the first entry of the vectors x's to be one to simplify the notation)

$$X = \begin{bmatrix} ---x^{(1)}--- \\ ---x^{(2)}--- \\ \vdots \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \end{bmatrix}, Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \end{bmatrix}$$

- then,

$$J(\theta) = \frac{1}{2}\|X\theta - Y\|^2, \text{ and}$$

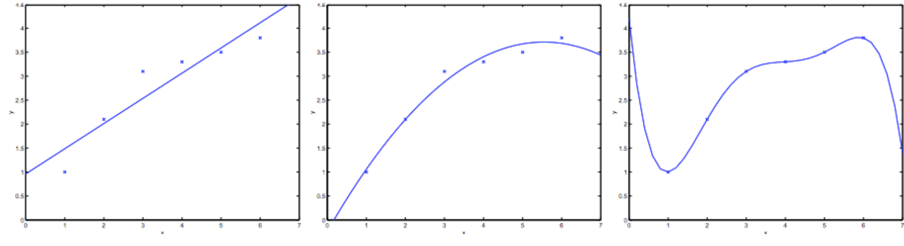
$$
\begin{aligned}
\nabla_\theta J(\theta) &= \frac{1}{2}\nabla_\theta \langle X\theta - Y, X\theta - Y\rangle \\
&= \frac{1}{2}\nabla_\theta \Big(\langle X\theta, X\theta\rangle - 2\langle \theta X, Y\rangle + \langle Y, Y\rangle\Big) \\
&= \frac{1}{2}\nabla_\theta \Big(\langle X^T X, \theta\theta^T\rangle - 2\langle X^T Y, \theta\rangle\Big) \\
&= X^T X\theta - X^T Y
\end{aligned}
$$

- here we used the fact that

$$\nabla_\theta \langle v, \theta\rangle = v \text{ and } \nabla_\theta \langle M, \theta\theta^T\rangle = 2M\theta$$

- then the global minimum θ^* can be directly computed by setting the gradient to zero

$$\theta^* = (X^T X)^{-1} X^T Y$$



- the left figure is linear regression of the form $h_\theta(x) = \theta_0 + \theta_1 x$
- if we add an extra feature of x^2 such that $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$, then we get a better model in the middle
- the right figure is using up to degree-5 polynomials $h_\theta(x) = \sum_{j=0}^{5} \theta_j x^j$

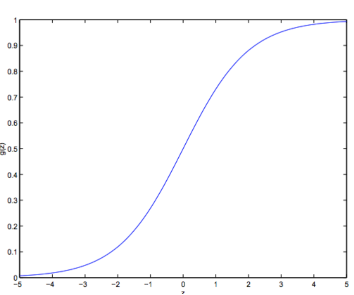
- **underfitting** (left): hypothesis class cannot capture the structure of the data
- **overfitting** (right): hypothesis class memorizes the training data and cannot generalize to new samples
- the choice of features is important to avoid under/over fitting

Logistic regression

- consider the task of "spam filter" that determines whether a given email is a spam or not
- we focus on **binary classification** tasks, where $y \in \{0, 1\}$
- such discrete valued output variable is called a **label**

- a **logistic function** or the **sigmoid function** is

$$g(z) = \frac{1}{1 + e^{-z}}$$



- logistic regression uses the parametric function class

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

- other choices of $g(\cdot)$ that is bounded between zero and one can be used, but logistic function is a natural one (for many reasons)
- one property: derivative

$$\begin{aligned} g'(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\ &= \frac{1}{(1+e^{-z})^2} e^{-z} \\ &= \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) \\ &= g(z)(1-g(z)) \end{aligned}$$

- we need a proper loss function $\ell(h_\theta(x), y)$ to define a cost function

- we make this formal by formulating it as a maximum likelihood problem for a natural probabilistic model:

$$P(y=1|x,\theta) \;:=\; h_\theta(x)\,, \quad P(y=0|x,\theta) \;:=\; 1-h_\theta(x)$$

- this can be written as

$$P(y|x,\theta) = h_\theta(x)^y(1-h_\theta(x))^{1-y}$$

- assuming n samples are generated i.i.d. the log-likelihood of observing those samples is

$$\begin{aligned} L(\theta) &= \log\Big(\prod_{i=1}^n P(y^{(i)}|x^{(i)},\theta)\Big) \\ &= \sum_{i=1}^n \log\Big(h_\theta(x^{(i)})^{y^{(i)}}(1-h_\theta(x^{(i)}))^{1-y^{(i)}}\Big) \\ &= \sum_{i=1}^n \Big(y^{(i)}\log h_\theta(x^{(i)}) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\Big) \end{aligned}$$

- now we can use the standard approach of using gradient ascent to maximize the log-likelihood

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} L(\theta) &= \sum_{i=1}^{n} \left(y^{(i)} \frac{1}{g(\theta^T x^{(i)})} - (1 - y^{(i)}) \frac{1}{1 - g(\theta^T x^{(i)})} \right) \frac{\partial}{\partial \theta_j} g(\theta^T x^{(i)}) \\
&= \sum_{i=1}^{n} \left(\frac{y^{(i)}}{g(\theta^T x^{(i)})} - \frac{1 - y^{(i)}}{1 - g(\theta^T x^{(i)})} \right) g(\theta^T x^{(i)})(1 - g(\theta^T x^{(i)})) \frac{\partial}{\partial \theta_j} \theta^T x^{(i)} \\
&= \sum_{i=1}^{n} \left(y^{(i)}(1 - g(\theta^T x^{(i)})) - (1 - y^{(i)}) g(\theta^T x^{(i)}) \right) x_j^{(i)} \\
&= (y - h_\theta(x)) x_j
\end{aligned}$$

- which gives

$$\theta_j \leftarrow \theta_j + \alpha \sum_{i=1}^{n} (y^{(i)} - h_\theta(x^{(i)})) x_j^{(i)}$$

If the target $y^{(i)} = 1$ and the current estimate $h_\theta(x^{(i)}) = 0.5$ and $x_j^{(i)} = 2$, then we would want to increase θ_j such that $h_\theta(x^{(i)})$ increases. Gradient descent turns this intuition into a concrete learning algorithm.

Softmax regression

- consider multi-class classification problem with $y \in \{1, \dots, k\}$
- we use multinomial distribution of the form

$$P(y = a | x, \theta) = \frac{e^{\theta_a^T x}}{\sum_{b=1}^{k} e^{\theta_b^T x}}$$

  where the parameters are $\theta_a \in \mathbb{R}^{d_x+1}$ for $a \in \{1, \dots, k\}$
- in this **softmax regression** problem, our (vector-valued) function output is

$$h_\theta(x) = \begin{bmatrix} \frac{e^{\theta_1^T x}}{\sum_{b=1}^{k} e^{\theta_b^T x}} \\ \vdots \\ \frac{e^{\theta_k^T x}}{\sum_{b=1}^{k} e^{\theta_b^T x}} \end{bmatrix}$$

  estimating the probability that $y = a$ given x and θ.
- for parameter fitting, we can apply maximum likelihood to define a cost function and apply gradient ascent

Battle against overfitting: cross validation and regularization

- Consider using a polynomial features of the form

$$h_\theta(x) = g(\theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_k x^k)$$

  and wish to decide the optimal k for your problem
- consider a case where we have a finite set of models $\mathcal{M} = \{M_1, \cdots, M_d\}$
- for example, M_i could be i-th order polynomial regression model
- here is a recipe for making such decisions
  - Train each M_i model on training data S and get a predictor h_i
  - Pick the predictor with the smallest training error
- this fails miserably, as in the polynomial example, higher-order polynomials will generate predictors that fits better and have smaller training error
- however, it leads to over-fitting with high variance
- **hold-out cross validation** works better

- Cross validation
  1. randomly split given data $S = \{(x^{(i)}, y^{(i)}))\}_{i=1}^{n}$ into S_{train} (say 70%) and S_{CV}
  2. train each model M_j on S_{train} to get a predictor h_j for each model class M_j
  3. select the hypothesis h_j that has the smallest error $E_{S_{\text{CV}}}(h_j)$ on the hold out cross validation set S_{CV} where

$$E_{S_{\text{CV}}}(h_j) = \frac{1}{|S_{\text{CV}}|} \sum_{i \in S_{\text{CV}}} \ell(h_j(x^{(i)}), y^{(i)})$$

- this ensures we get a better estimate of the generalization error (the error on the unseen feature x)
- typical choice of hold out set size is 30$\sim$40%
- one problem with hold out cross validation is that we are only testing models that are trained on 70% of data, which leads to k-**fold cross validation**

- **k-fold cross validation**
  1. randomly split the data $S = \{(x^{(i)}, y^{(i)}))\}_{i=1}^{n}$ into K disjoint subsets of n/K examples each, and call them $S_1, \dots, S_K$
  2. For each model M_j
     - ★ For each $k = 1, \dots, K$
     - ★ train the model M_j on $S_1 \cup S_2 cup \cdots \cup S_{k-1} \cup S_{k+1} \cup \cdots S_K$ to get a hypothesis h_{jk}
     - ★ test the hypothesis h_{jk} on S_k to get $E_{S_k}(h_{jk})$

     the estimated generalization error for model M_j is the average of $E_{S_k}(h_{jk})$'s
  3. pick the model with the lowest estimated generalization error, and retrain the model on the entire S

- typical choice of folds is $K = 10$
- when $K = n$, we are leaving out only one sample per experiment, which is called **leave-one-out cross validation**

Regularization

- Parameter fitting using Maximum Likelihood (ML):

$$\theta_{\text{ML}} \;=\; \arg\max_{\theta} \prod_{i=1}^{n} P(y^{(i)}|x^{(i)}, \theta)$$

- here, we are taking a **frequentist** view and take the unknown θ as a static or **deterministic** quantity that does not change
- an alternative view of parameter estimation is **Bayesian** view, where θ is also a **random variable** with a **prior distribution** $P(\theta)$ on θ that expresses our prior beliefs about the parameter
- if we have such a prior distribution (and we know it), then the posterior distribution on the parameter θ given the samples $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$ is (according to the Bayes rule)

$$\begin{aligned} P(\theta|S) &= \frac{P(S|\theta)P(\theta)}{P(S)} \\ &= \frac{P(\theta)\prod_{i=1}^{n} P(y^{(i)}|x^{(i)}, \theta)}{\int_{\tilde{\theta}} \Big(P(\tilde{\theta}) \prod_{i=1}^{n} P(y^{(i)}|x^{(i)}, \tilde{\theta})\Big)\, d\tilde{\theta}} \end{aligned}$$

$$P(\theta|S) \;=\; \frac{P(\theta)\prod_{i=1}^{n} P(y^{(i)}|x^{(i)},\theta)}{\int_{\tilde\theta}\Big(\,P(\tilde\theta)\,\prod_{i=1}^{n} P(y^{(i)}|x^{(i)},\tilde\theta)\Big)\,d\tilde\theta}$$

- $P(\theta)$ is a given prior distribution, say Gaussian

$$P(\theta) = \frac{1}{(2\pi|\Sigma|)^{1/2}}\, e^{-\frac{1}{2}(\theta-\mu)^T\Sigma^{-1}(\theta-\mu)}$$

- $P(y^{(i)}|x^{(i)},\theta)$ is whatever model we are using, say logistic regression

$$P(y|x,\theta) = \Big(\frac{1}{1+e^{-\theta^T x}}\Big)^{y}\Big(1-\frac{1}{1+e^{-\theta^T x}}\Big)^{1-y}$$

- if we can compute $P(\theta|S)$, then we can solve any prediction/estimation tasks, for example predicting y from x:

$$P(y|x,S) = \int_\theta P(y|x,\theta)P(\theta|S)\,d\theta$$

- unfortunately, it is computationally very hard to compute $P(\theta|S)$ as it involves integration that does not typically give a closed-form solution

- instead, a **MAP (maximum a posteriori)** estimate of θ is commonly used, given by

$$\theta_{\text{MAP}} \;=\; \arg\max_{\theta} \prod_{i=1}^{n} P(y^{(i)}|x^{(i)}, \theta) P(\theta)$$

  the only difference compared to ML estimate is the prior $P(\theta)$
- in practical applications, a common choice is a Gaussian distribution $\theta \sim N(0, \lambda \mathbb{I})$, which gives

$$\theta_{\text{MAP}} \;=\; \arg\min_{\theta} \sum_{i=1}^{n} -\log P(y^{(i)}|x^{(i)}, \theta) \;+\; \lambda \|\theta\|^2$$

- this encourages small norm solution (compared to ML), which causes MAP estimate to be less susceptible to overfitting than ML estimates
- this type of regularization is also referred to as **weight decay**

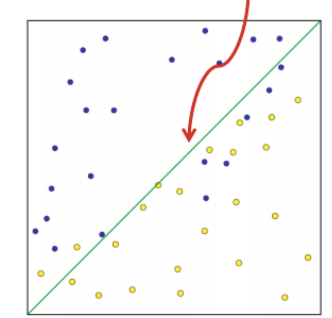
The goal of adding regularizer is to encourage a model that is less sensitive to how test data differ from training data. For example, if your model is $h_{\theta}(x) = \theta^T x$, then θ_j large implies that the model is sensitive to the variations in x_j. A model with small θ's are less sensitive to variations in the test data.

Logistic regression is a **linear** classifier

$$\begin{aligned}\theta_{\text{logistic}} &= \arg\max_{\theta} L(\theta) \\ &= \arg\max_{\theta} \sum_{i=1}^{n} \left(y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)})) \right)\end{aligned}$$

$$h_{\theta}(x) = P(y = 1|x) = \frac{1}{1 + e^{-\theta^T x}}$$

linear decision boundary at $\theta^T x = 0$

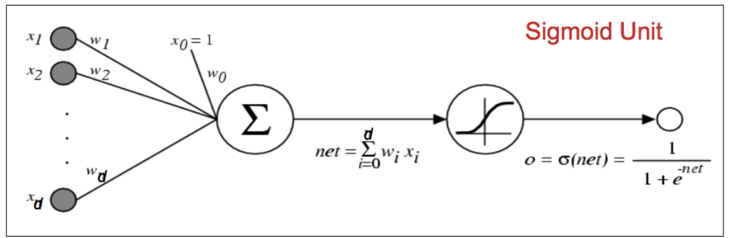


- decision boundary:
  $P(y = 1|x) = P(y = 0|x)$
  at $\theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$

Logistic function as a single layer network

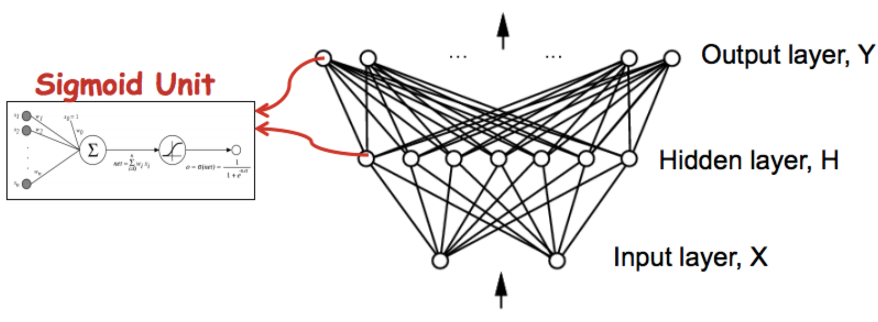
- network representation of a function (e.g. logistic regression)
- we will use w (instead of θ) for the weights as it is more standard in the neural network community
- output of a network $o(x)$

$$o(x) = g(w^T x) = \frac{1}{1 + e^{-w^T x}}$$

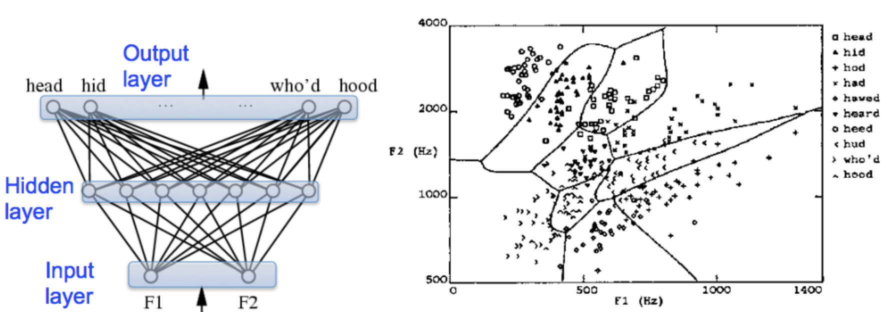


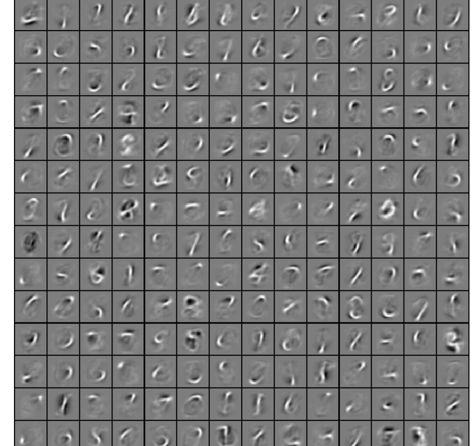
Deep Neural Network

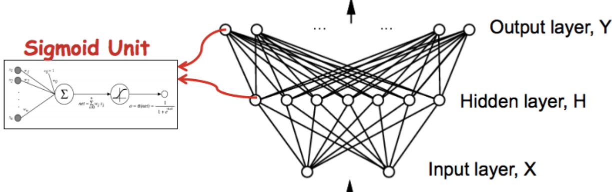
- under practical scenarios, we want to learn **non-linear** decision boundaries
- **Neural networks** are a parametric family of functions $f_W(x) : \mathbb{R}^d \to \mathcal{Y}$ represented by network of logistic units



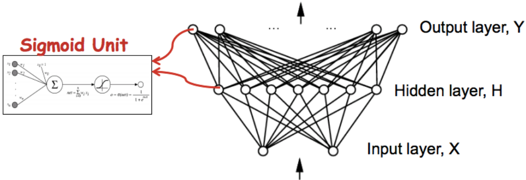
- Example: 2-layer neural network trained to distinguish vowel sounds using 2 formants (features)
- a highly non-linear decision boundary can be learned from 2-layer neural networks






Sigmoid Unit
Output layer, Y
Hidden layer, H
Input layer, X

Prediction (with an already learned model)



- for prediction using (learned) neural network, **forward propagation** is used
- start from input layer and compute at each subsequent layer the output of the sigmoid unit
- first layer:

$$o_{1j}(x) \;=\; g((w_1^j)^T x)$$

- second layer:

$$o_{2j}(x) \;=\; g\Big(\,(w_2^j)^T g((w_1)^T x)\,\Big)$$

where $g((w_1)^T x) = [g((w_1^1)^T x);\ \cdots\ ;\ g((w_1^k)^T x)]$ is a vector of outputs from 1st layer with k hidden units

Training a model from data

- regression with neural network, solves for

$$w^* = \arg\min_w \underbrace{\frac{1}{2}\sum_{i=1}^{n} \left\| y^{(i)} - h_w(x^{(i)}) \right\|^2}_{J(w)}$$

  in the case of a quadratic loss of $\ell(y, h_w(x)) = (1/2)\|y - h_w(x)\|^2$
- note that $h_w(x)$ is not a convex function in w
- nevertheless the standard approach is to apply **gradient descent** to find a good minimum w
- **backpropagation** is used to evaluate the gradient

- some calculus background to derive backpropagation
- consider composition of multiple functions

$$z_1 = f_1(x)\ , \quad z_2 = f_2(x) \text{ and } y = g(z_1, z_2) = g(f_1(x), f_2(x))$$

then

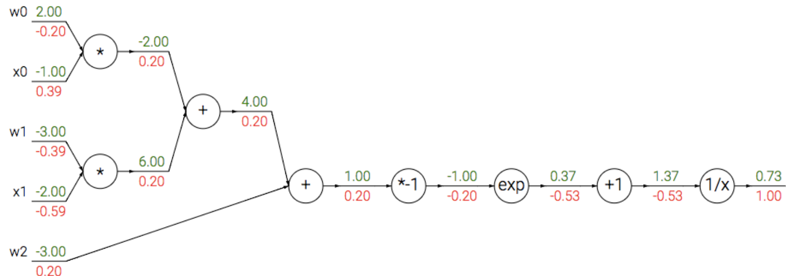
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z_1}\frac{\partial z_1}{\partial x} + \frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial x}$$

- we want to first compute the derivative w.r.t. the **output units** $\frac{\partial J(w)}{\partial w_{2,j_1 j_2}}$

- main idea behind **backpropagation** is composition
- as an example consider the task of computing the gradient of $f_w(x)$ with respect to w for

$$f_w(x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

- we can represent $f_w(x)$ as a network of computation neurons:

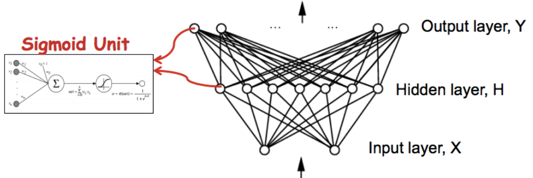


green shows forward propagation to compute $f_w(x)$
red shows backpropagation to compute $\frac{\partial f_w(x)}{\partial w}$

- backpropagation
  - start with 1.00 and propagate backward

1. $1.00 * (-1/x^2) = 1.00 * (-1/1.37^2) = -0.53$
2. $-0.53 * (1) = -0.53$
3. $-0.53 * \exp(x) = -0.53 * \exp(-1) = -0.195 \simeq -0.2$
4. $-0.2 * -1 = 0.2$
5. $-0.2 * 1 = -0.2$
6. $\frac{\partial f}{\partial w_0} = 0.2 * x_0 = -0.2$
7. $\frac{\partial f}{\partial w_1} = 0.2 * x_1 = -0.4 (\simeq -0.39)$
8. $\frac{\partial f}{\partial w_2} = 0.2$

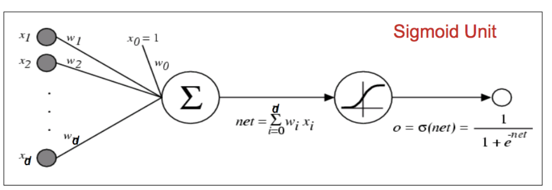
Output layer



- compute $\frac{\partial J(w)}{\partial w_{2,j_1 j_2}}$ for $J(w) = \frac{1}{2}\sum_{j_2}\left(y_{j_2}^{(i)} - o_{2j_2}(x^{(i)})\right)^2$ where $o_{2j_2}(x^{(i)}) = g(\sum_{j_1} w_{2,j_1 j_2} o_{1j_1}(x^{(i)}))$

$$\frac{\partial J(w)}{\partial w_{2,j_1 j_2}} = \underbrace{\frac{\partial J(w)}{\partial o_{2j_2}(x^{(i)})}}_{(o_{2j_2}(x^{(i)}) - y_{j_2}^{(i)})} \times \underbrace{\frac{\partial o_{2j_2}(x^{(i)})}{\partial w_{2,j_1 j_2}}}_{o_{2j_2}(x^{(i)})(1-o_{2j_2}(x^{(i)}))o_{1j_1}(x^{(i)})}$$

If the target $y_{j_2}^{(i)} = 1$ and $o_{2j_2}(x^{(i)}) = 0.5$ and $o_{1j_1}(x_j^{(i)}) = 2$, then we would want to increase $w_{2,j_1 j_2}$ such that $o_{2j_2}(x^{(i)})$ increases.

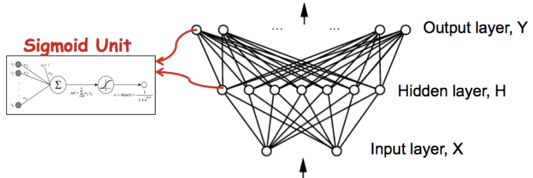


- the gradient for a single sigmoid unit is

$$\frac{\partial}{\partial w_j} J(w) = -\sum_{i=1}^{n} (y^{(i)} - h_w(x^{(i)})) h_w(x^{(i)}) (1 - h_w(x^{(i)})) x_j^{(i)}$$

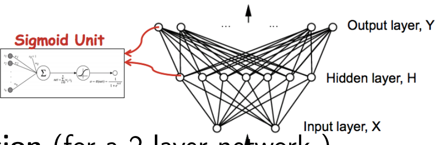
- derivation:

Hidden layer



- compute $\frac{\partial J(w)}{\partial w_{1,j_0 j_1}}$ for $J(w) = \frac{1}{2}\sum_{j_2}\left(y_{j_2}^{(i)} - o_{2j_2}(x^{(i)})\right)^2$ where $o_{2j_2}(x^{(i)}) = g(\sum_{j_1} w_{2,j_1 j_2} o_{1j_1}(x^{(i)}))$, and $o_{1j_1}(x^{(i)}) = g(\sum_{j_0} w_{1,j_0 j_1} x_{j_0}^{(i)})$

$$\frac{\partial J(w)}{\partial w_{1,j_0 j_1}} = \sum_{j_2} \underbrace{\frac{\partial J(w)}{\partial o_{2j_2}(x^{(i)})}}_{(o_{2j_2} - y_{j_2}^{(i)})} \times \underbrace{\frac{\partial o_{2j_2}(x^{(i)})}{\partial o_{1j_1}}(x^{(i)})}_{o_{2j_2}(1-o_{2j_2})w_{2,j_1 j_2}} \times \underbrace{\frac{\partial o_{1j_1}(x^{(i)})}{\partial w_{1,j_0 j_1}}}_{o_{1j_1}(1-o_{1j_1})x_{j_0}^{(i)}}$$



- **backpropagation** (for a 2-layer network)
  - initialize w randomly
  - Repeat until convergence
  - For each training example $(x^{(i)}, y^{(i)})$
  - Compute $\hat{y}^{(i)} \leftarrow o_2(x^{(i)})$
  - For each output unit k

$$\delta_k^{(i)} \leftarrow \hat{y}_k^{(i)}(1 - \hat{y}_k^{(i)})(y_k^{(i)} - \hat{y}_k^{(i)})$$

  - Compute $o_{1,j}^{(i)} \leftarrow g((w_{0,j})^T x^{(i)})$ for all hidden units j
  - For each hidden unit j

$$\epsilon_j^{(i)} \leftarrow o_{1,j}^{(i)}(1 - o_{1,j}^{(i)}) w_{1,j}^T \delta^{(i)}$$

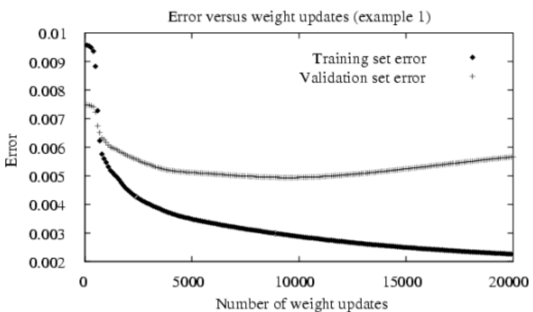
  - Update each weight as

$$\begin{aligned} w_{2,jk} &\leftarrow w_{2,jk} + \alpha \delta_k^{(i)} o_{1,j}^{(i)} \\ w_{1,\ell j} &\leftarrow w_{1,\ell j} + \alpha \epsilon_j^{(i)} x_\ell^{(i)} \end{aligned}$$

- backpropagation
  - gradient descent on all weights
  - easily generalizes to any directed network
  - due to non-convexity, will find local minima in general
  - depends on the initialization
    might require multiple trials
  - can include weight **momentum**

$$\Delta w_{2,jk}(t+1) \leftarrow \alpha \delta_k^{(i)} o_{1,j}^{(i)} + \beta \Delta w_{2,jk}(t)$$

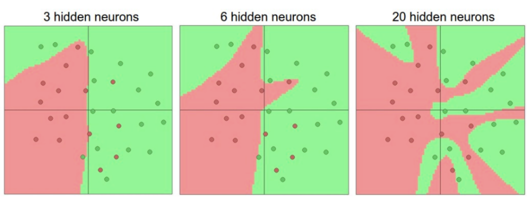
  - minimizes error over training examples
    might not generalize to test data
  - training takes hundreds of thousands of iterations
    typical training is very slow
  - once trained, testing is fast
  - termination criteria: increase in validation set error



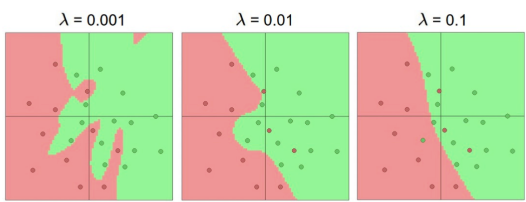
- handling **overfitting**
  as number of iterations increases, training error decreases but the validation error starts to increase
- how do we know when to stop?
- common techniques: cross-validation, regularization, controlling the size of the network

Representation power vs. size and regularization

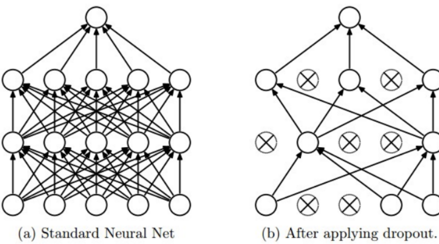
- more layers and more nodes in each layer gives larger representation power, but can lead to overfitting



- larger regularization coefficient gives smoother surface, potentially avoiding overfitting



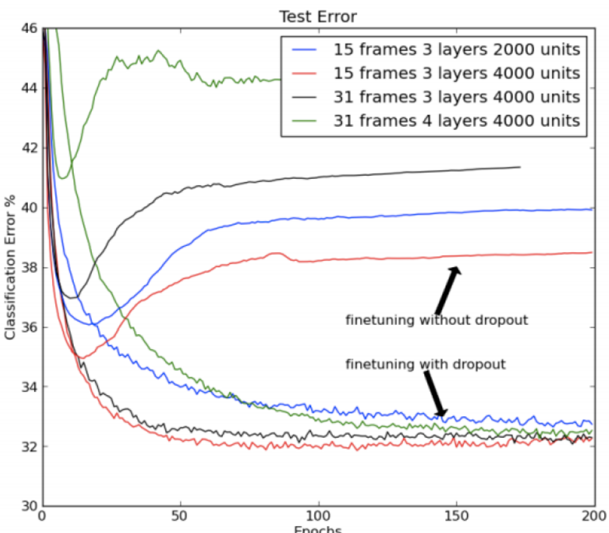
Dropout (yet another regularization technique)



- **Dropout** is another recently introduced (["Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Srivastava, Hinton, Krizhevsky, Sutskever, Salakhutdinov, 2014]) technique for regularization
- at training, each "neuron" is active with some probability p, and set to zero otherwise
- at testing, all neurons are active, but scaled by p

- pseudo code

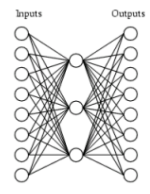
```
p = 0.5
# probability of keeping a unit active.
def train_step(X):
      # forward pass for example 2-layer neural network
      H1 = np.maximum(0, np.dot(W1, X) + b1)
      U1 = np.random.rand(*H1.shape) < p
      # first dropout mask
      H1 *= U1
      # drop
      out = np.dot(W2, H1) + b2
      # backward pass:  compute gradients...  (not shown)
       ⋮
      # perform parameter update...  (not shown)
       ⋮
def predict(X):
      # ensembled forward pass
      H1 = np.maximum(0, np.dot(W1, X) + b1) * p
      # NOTE: scale the activations
      out = np.dot(W2, H1) + b2
```



50% dropout for hidden layer and 20% dropout for input layer

Representation power

- representation power of neural network

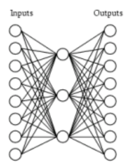


A target function:

Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

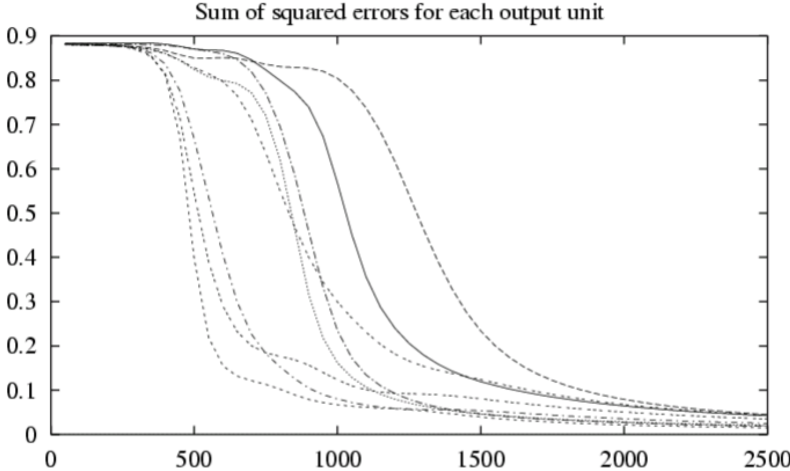
can this be learned?

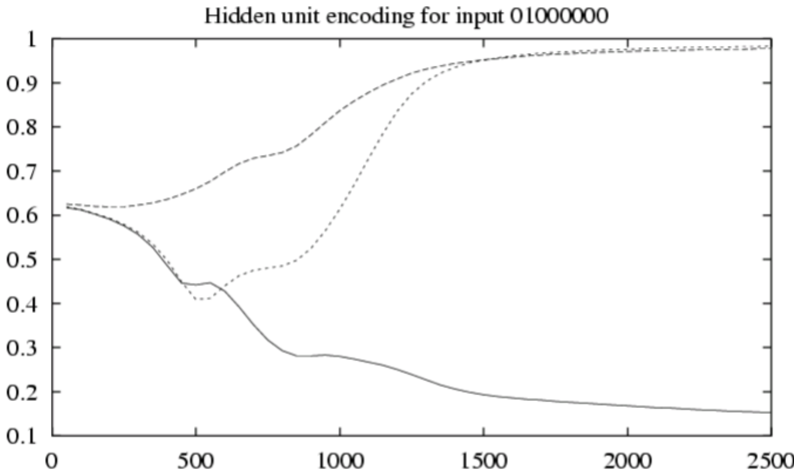
A network:



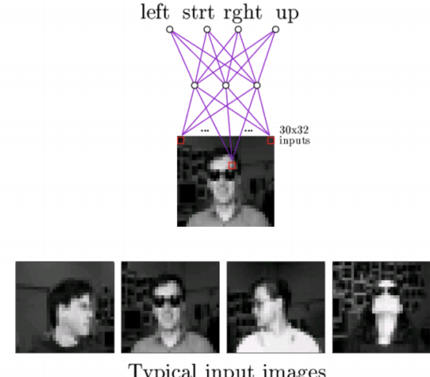
Learned hidden layer representation:

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001


Sum of squared errors for each output unit
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
500
1000
1500
2000
2500


Hidden unit encoding for input 01000000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
500
1000
1500
2000
2500

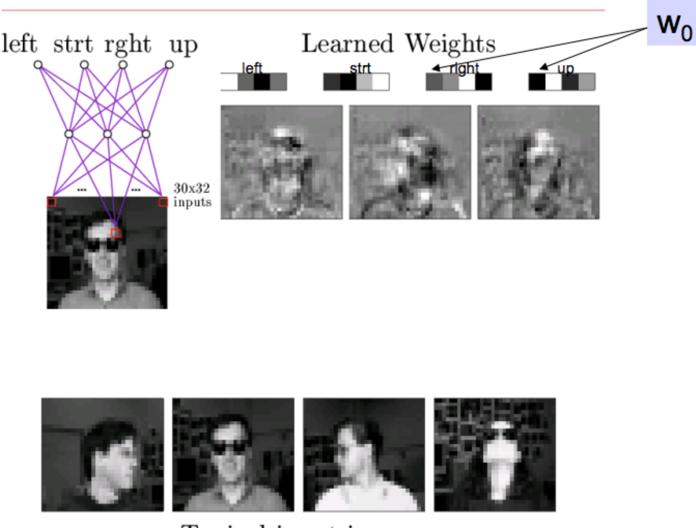
- face recognition



Typical input images

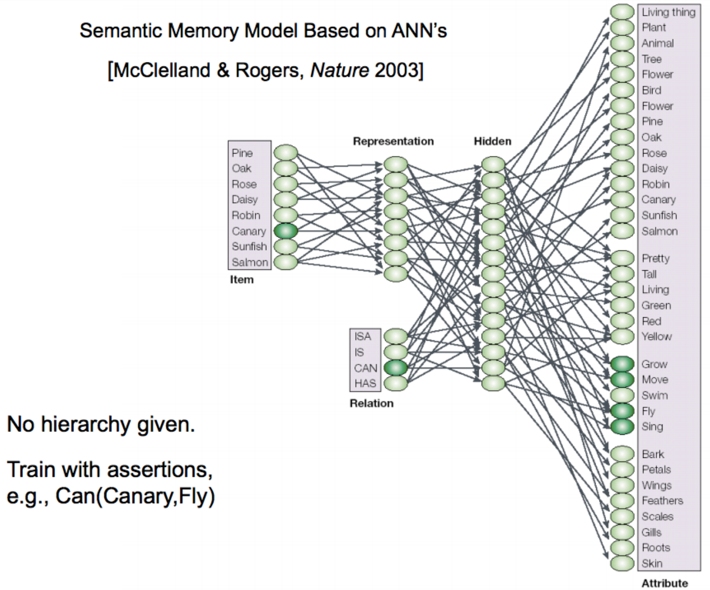
90% accuracy in learning the head pose

- face recognition



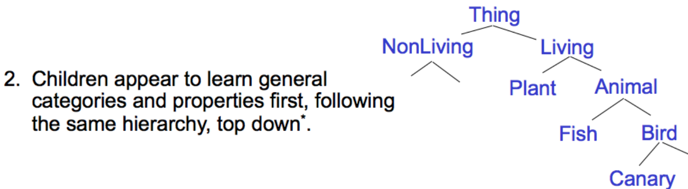
Typical input images

- semantic memory



- semantic memory

1. Victims of Semantic Dementia progressively lose knowledge of objects
   But they lose specific details first, general properties later, suggesting hierarchical memory organization



- semantic memory

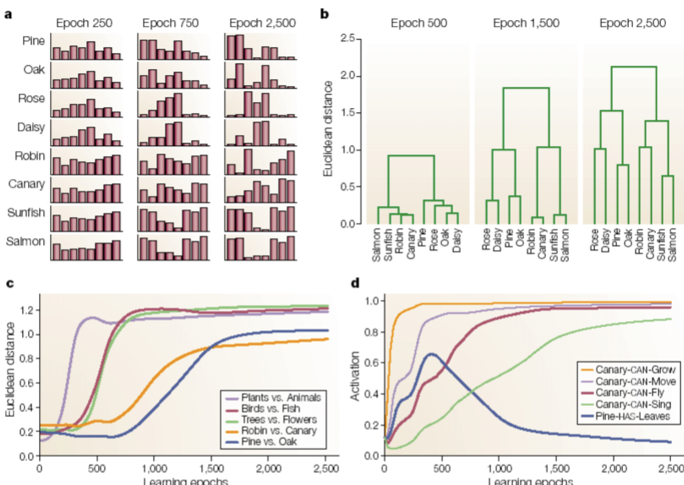


Figure 4 | **The process of differentiation of conceptual representations.** The representations are those seen in the feedforward network model shown in FIG. 3. **a** | Acquired patterns of activation that represent the eight objects in the training set at three points in the learning process (epochs 250, 750 and 2,500). Early in learning, the patterns are undifferentiated; the first difference to appear is between plants and animals. Later, the patterns show clear differentiation at both the superordinate (plant–animal) and intermediate (bird–fish/tree–flower) levels. Finally, the individual concepts are differentiated, but the overall hierarchical organization of the similarity structure remains. **b** | A standard hierarchical clustering analysis program has been used to visualize the similarity structure in the

Deep learning

- the goal of deep learning is to capture the hierarchy of features that capture increasing level of abstraction/summarization

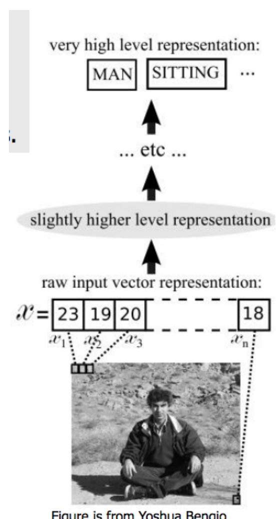


Figure is from Yoshua Bengio

- biological motivation: mammal brain is organized in a deep architecture
- (brief) history of deep learning
  - researchers strived to train deep multi-layer neural network for decades without success
  - no successful attempt reported until 2006 (hard to train deep models, and leads to poorer results)
  - exception: convolutional neural networks, Le Cun 1998
  - Support Vector Machine (SVM) from 1993 is a shallow architecture, that works better than any deep architectures in 1990's
  - led to many researchers abandoning deep learning
  - breakthrough in 2006
    [A Fast Learning Algorithm for Deep Belief Networks, 2006, Hinton, Osindero, Teh]
    [Greedy Layer-wise Training of Deep Networks, 2007, Bengio, Lamblin, Popovici, Larochelle]

Convolutional Neural Network

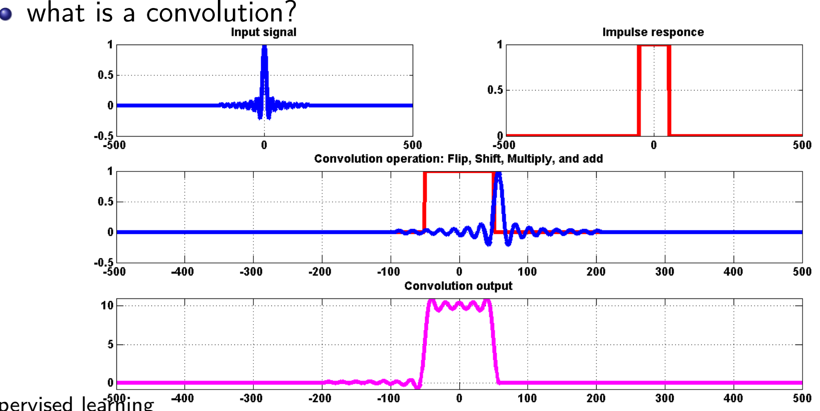
- **Convolutional Neural Networks** are very similar to Neural Networks
- the whole network takes as input X and minimizes loss w.r.t. Y
- the main difference is that it makes the explicit assumption that the input is an **image**, that allows to explicitly impose some structures to the neural network architecture, such that we reduce the number of parameters dramatically
- regular neural networks do not scale well to full images: CIFAR-10 images are only $32 \times 32 \times 3$ and a single neuron at the first layer will have $32 \times 32 \times 3 = 3072$ weights

- this gets worse for moderate size images as $200 \times 200 \times 3 = 120,000$ weights are needed for a single neuron

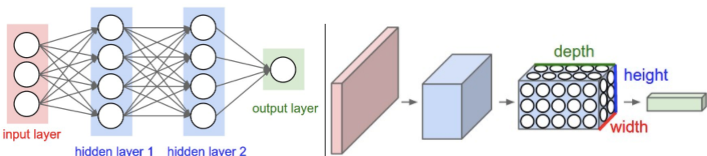
- as the dimension $d = 120,000$ is comparable to the sample size $n = 60,000$, i.e. $n \simeq d$
- and we want to have several neurons and several layers
- too many parameters quickly lead to overfitting

- main idea: to take advantage of scale, shift, rotation invariance of images

- what is a convolution?

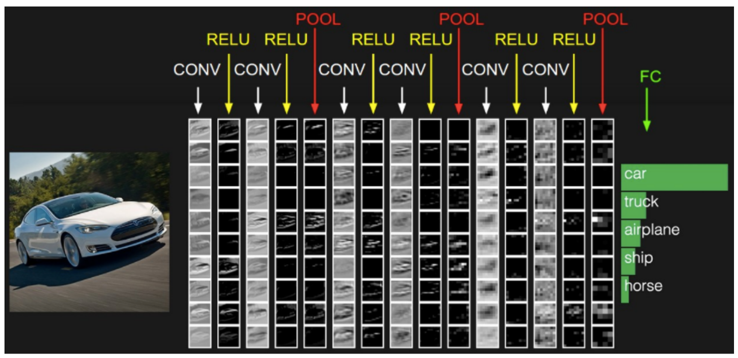


- **3D volumes of neurons:**
  - each layer in a ConvNet is arranged in 3-dimensions: **width, height, depth**
  - we use depth to refer to the depth of the volume of 3-d neurons in a single layer (and not the number of layers)
  - for example, the CIFAR-10 input image has dimensions $32 \times 32 \times 3$ (width, height, depth)
  - instead of each neuron **fully connected** to the previous layer, ConvNet neurons will have connection to a small **window** of region
  - final output layer will have dimension $1 \times 1 \times 10$ for CIFAR-10

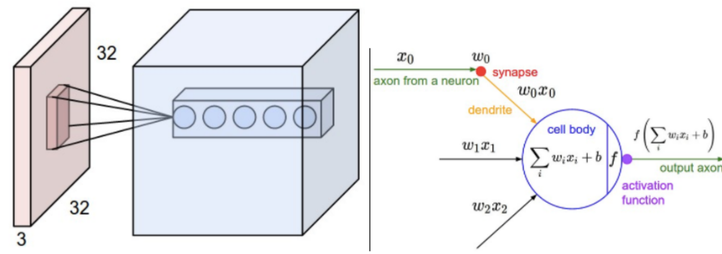


- Each layer of ConvNet transforms input volume to output volume via a differentiable function
- main ingredients are **Convolutional Layer, Pooling Layer, and Fully Connected Layer** stacked to build a ConvNet
- Example of (simple) ConvNet with [INPUT-CONV-RELU-POOL-FC]
  - INPUT $[32 \times 32 \times 3]$ for raw pixel RGB values
  - CONV layer $[32 \times 32 \times 12]$ will output inner product between a small window of input around each neuron and the weights, and we can choose to use 12 such filters
  - RELU $[32 \times 32 \times 12]$ applies entry-wise activation function of $\max(0, x)$ and the dimension does not change
  - POOL $[16 \times 16 \times 12]$ performs downsampling along the spatial dimension (width,height)
  - FC (Fully Connected) layer $[1 \times 1 \times 10]$ is typical neural network layer for outputing scores for the 10 categories in CIFAR-10

- ConvNets transform an image into 10-dimensional scores
- CONV/FC layers have weights to be trained via gradient descent
- RELU/POOL layers are fixed functions



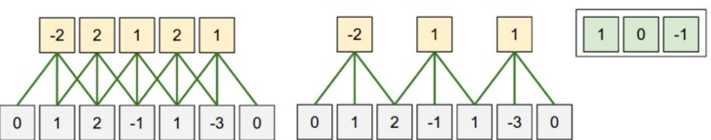
Convolutional layer



- example of INPUT[$32 \times 32 \times 3$] and CONV[$32 \times 32 \times 5$] layers
- each neuron is connected to a small region spatially (width,height), for example 3×3, but fully in depth
- multiple neurons (5 in the example) along the depth process the same region of input (this layer has 5 filters)
- the neuron performs standard operation on this region

$$\text{output} = \text{ReLU}\left(\sum_j w_j x_j\right)$$

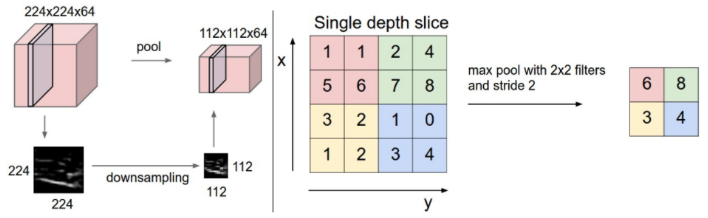
- intuition
  - forward pass: we are sliding 5 filters over the input image across width and height
  - each filter produces a 2-d output
  - we will learn the right filters to capture visual features such as edge of a certain orientation, or a batch of colors (at lower layer) and honeycomb or wheel patterns (at higher layer)
- local connectivity (in spatial dimension) allows us to keep the number of parameters small, specified by **depth, stride, and zero-padding**
  - **depth** is a hyper parameter specifying the number of filters
  - **stride** is the rate of moving the filter; if stride is 2 then the filter jumps 2 pixels at a time, resulting in half of the width and height at the output
  - **zero-padding** pads zeros around the border of the input to preserve the size at the output



- number of parameters is still large, even with local window
  real ConvNets have $11 \times 11 \times 3 = 363$ weights for 1 neuron and $55 \times 55 \times 96 = 290,400$ neurons, totaling $105,705,600$ parameters in the first layer
- **parameter sharing** dramatically reduces the number of parameters to learn
- since a good filter at one position in the image is also a good filter at other locations, we use the same weight for all windows
- the total number of parameters is now $96 \times 11 \times 11 \times 3 = 34,848$
- this is why the architecture is called a **convolutional network**

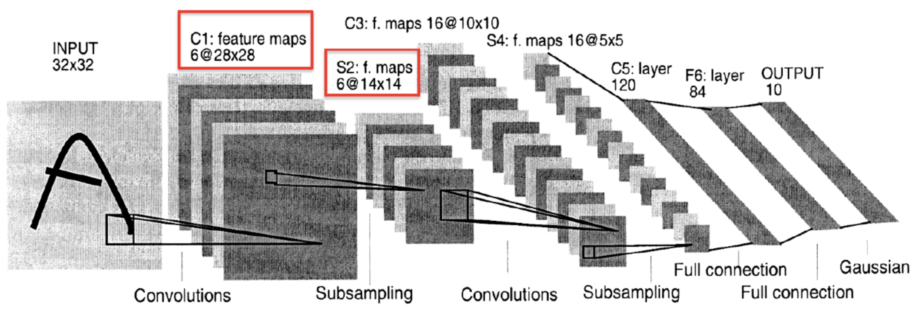
Pooling layer

- **pooling** progressively reduces the (spatial) size of the network
- reduces parameters and computation
- for example, a pooling neuron takes input a 2×2 region and outputs the MAX of these 4 numbers, moving across the input with a stride of 2
- the output width and height is now reduced by half
- this is a downsampling with non-linear functions
- other functionals include, **max pooling, average pooling, and L2-norm pooling**
- it should be a function independent of the input permutation



- pooling and weight sharing provides translation invariance
-
- however, there are attempts to get rid of pooling and only use CONV layers with larger strides
- getting rid of pooling is important for training generative models such as Variational AutoEncoders (VAE) and Generative Adversarial Networks (GAN)

LeNet, 1990's



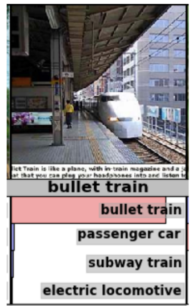
- 82 error made by LeNet

- 35 error made by Ciresan et al.
- further, most of the time the true answer is in the top-2 prediction
- idea: train with transformed samples

2 / 1 7	1 / 7 1	8 / 9 8	9 / 5 9	9 / 7 9	5 / 3 5	8 / 2 3
9 / 4 9	5 / 3 5	4 / 9 7	9 / 4 9	4 / 9 4	2 / 0 2	5 / 3 5
6 / 1 6	4 / 9 4	0 / 6 0	6 / 0 6	6 / 8 6	1 / 7 9	1 / 7 1
9 / 4 9	0 / 5 0	5 / 3 5	8 / 9 8	9 / 7 9	7 / 1 7	1 / 6 1
7 / 2 7	8 / 5 8	2 / 7 8	6 / 1 6	5 / 6 5	4 / 9 4	0 / 6 0

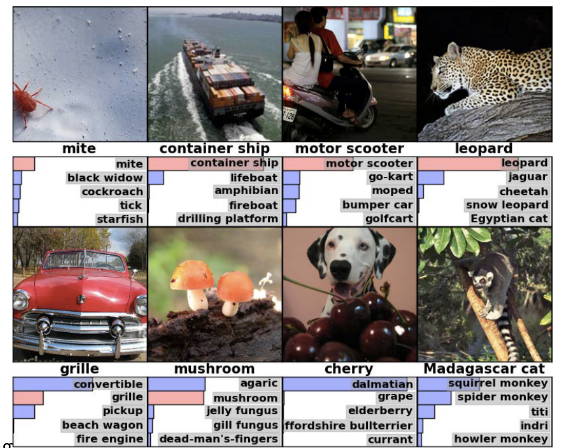
ILSVRC-2012 challenge on ImageNet

- 28×28 grey-scale to 256×256 color
- 10 classes to 1,000 classes
- multiple objects
- natural 3-d scene



winner: AlexNet

- Alex Krizhevsky, Ilya Sutskever and Geoff Hinton, 2012
- mirror image
- subsampling to get 224×224 patches from 256×256 images
- ReLU activation is faster to train and more expressive
- Dropout to regularize



- **ZF Net**: ILSVRC 2013 winner, Matthew Zeiler and Rob Fergus, parameter tuning over AlexNet
- **GoogLeNet**: ILSVRC 2014 winner, Szegedy et al., Inception Module,
- **VGGNet**: runner-up in ILSVRC 2014, Karen Simonyan and Andrew Zisserman, depth helps with 16 CONV/FC layers
- **ResNet**: ILSVRC 2015 winner, Kaiming He et al., skip connections and a heavy use of batch normalization

-