Administration

Questions

- Registration
- Hw3 is out
  - Due on Thursday 10/6
- Lecture Captioning (Extra-Credit)
  - Look at Piazza for details
- Scribing lectures
  - With pay; come talk to me/send email.

Projects

- Projects proposals are due on Friday 10/14/16
- Within a week we will give you an approval to continue with your project along with comments and/or a request to modify/augment/do a different project. There will also be a mechanism for peer comments.
- We encourage team projects – a team can be up to 3 people.
- Please start thinking and working on the project now.
- Your proposal is limited to 1-2 pages, but needs to include references and, ideally, some of the ideas you have developed in the direction of the project (maybe even some preliminary results).
- Any project that has a significant Machine Learning component is good.
- You can do experimental work, theoretical work, a combination of both or a critical survey of results in some specialized topic.
- The work *has* to include some reading. Even if you do not do a survey, you must read (at least) two related papers or book chapters and relate your work to it.
- Originality is not mandatory but is encouraged.
- Try to make it interesting!

Examples

- KDD Cup 2013:
  - "Author-Paper Identification": given an author and a small set of papers, we are asked to identify which papers are really written by the author.
    - https://www.kaggle.com/c/kdd-cup-2013-author-paper-identification-challenge
  - "Author Profiling": given a set of document, profile the author: identification, gender, native language, ….
- Caption Control: Is it gibberish? Spam? High quality text?
  - Adapt an NLP program to a new domain
- Work on making learned hypothesis (e.g., linear threshold functions, NN) more comprehensible
  - Explain the prediction
- Develop a (multi-modal) People Identifier
- Compare Regularization methods: e.g., Winnow vs. L1 Regularization
- Large scale clustering of documents + name the cluster
- Deep Networks: convert a state of the art NLP program to a deep network, efficient, architecture.
- Try to prove something

Neural Networks

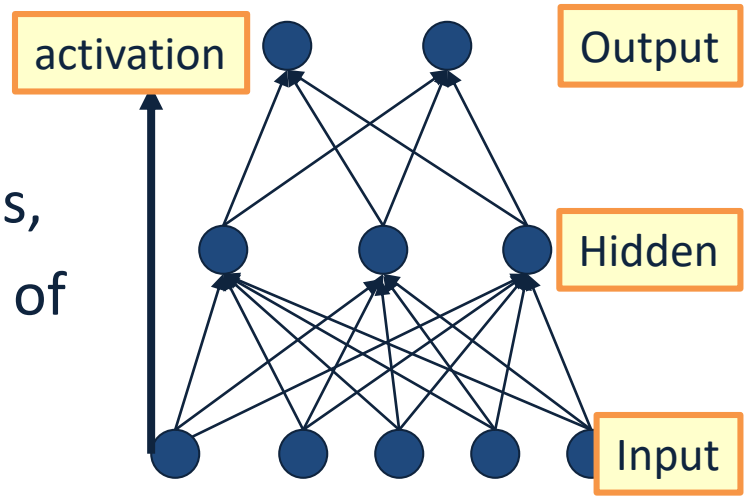
- **Robust** approach to approximating **real-valued**, **discrete-valued** and **vector valued** target functions.
- Among the most effective **general purpose** supervised learning method currently known.
- Effective especially for **complex and hard to interpret input data** such as real-world sensory data, where a lot of supervision is available.
- The **Backpropagation algorithm** for neural networks has been shown successful in many practical problems
  - handwritten character recognition, speech recognition, object recognition, some NLP problems

Neural Networks

- Neural Networks are **functions**: $\mathrm{NN}: X \rightarrow Y$
  - where $X = [0,1]^n$, *or* $\{0,1\}^n$ *and* $Y = [0,1]$, $\{0,1\}$
- NN can be used as an approximation of a target classifier
  - In their general form, even with a single hidden layer, NN can approximate any function
  - Algorithms exist that can learn a NN representation from labeled training data (e.g., Backpropagation).

Multi-Layer Neural Networks

- Multi-layer network were designed to overcome the computational (**expressivity**) limitation of a single threshold element.
- The idea is to **stack** several layers of threshold elements, each layer using the output of the previous layer as input.



Motivation for Neural Networks

- Inspired by **biological systems**
  - **But don't take this (as well as any other words in the new on "emergence" of intelligent behavior) seriously;**
- We are currently on rising part of a wave of interest in NN architectures, after a long downtime from the mid-90-ies.
  - Better computer architecture (GPUs, parallelism)
  - A lot more data than before; in many domains, supervision is available.
- Current surge of interest has seen very minimal algorithmic changes

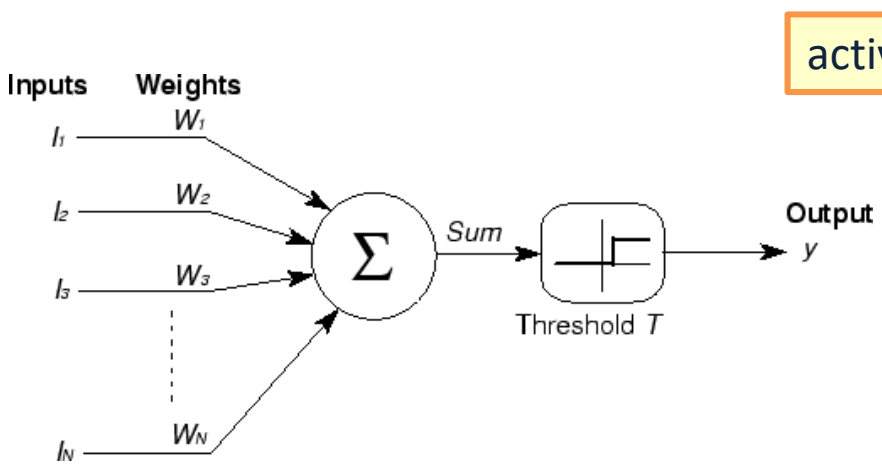
Motivation for Neural Networks

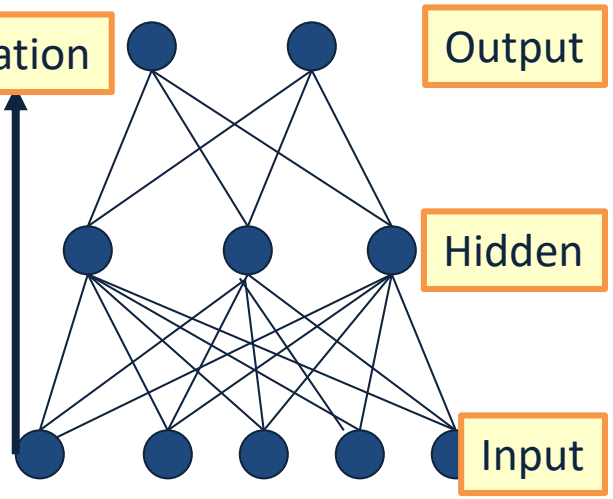
- Minimal to no algorithmic changes
- One potentially interesting perspective:
  - Before we looked at NN only as function approximators.
  - Now, we look at the intermediate representations generated while learning as meaningful
  - Ideas are being developed on the value of these intermediate representations for transfer learning etc.

- We will present in the next two lectures a few of the basic architectures and learning algorithms, and provide some examples for applications

Basic Unit in Multi-Layer Neural Network

- **Linear Unit**: $o_j = \vec{w}.\vec{x}$ multiple layers of linear functions produce linear functions. We want to represent nonlinear functions.
- **Threshold units:** $o_j = sgn(\vec{w}.\vec{x} - T)$ are not differentiable, hence unsuitable for gradient descent

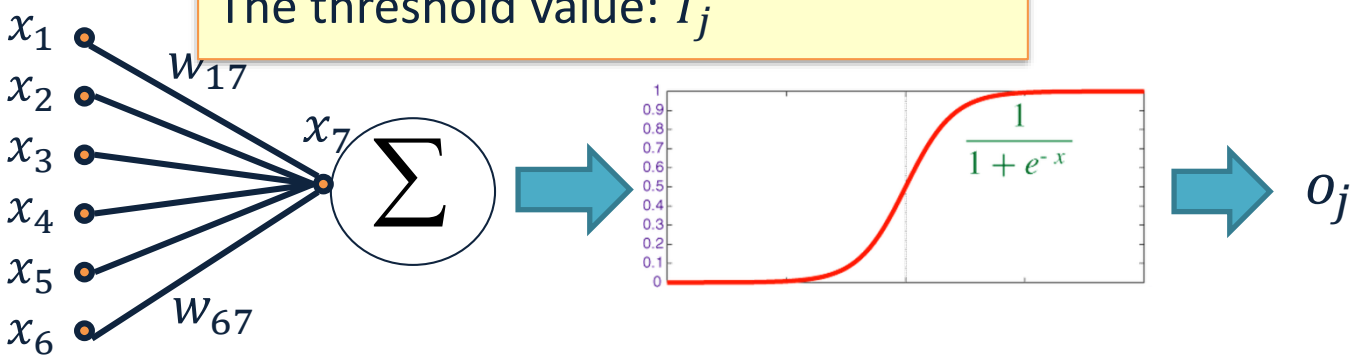




Model Neuron (Logistic)

- Neuron is modeled by a unit j connected by weighted

The parameters so far?
The set of connective weights: w_{ij}
The threshold value: T_j



Neuron Definition

- Use a non-linear, differentiable output function such as the sigmoid or logistic function
- Net input to a unit is defined as: $\text{net}_j = \sum w_{ij}.x_i$
- Output of a unit is defined as: $o_j = \frac{1}{1+\exp\left(-(\text{net}_j - T_j)\right)}$

History: Neural Computation

- McCollough and Pitts (1943) showed how linear threshold units can be used to compute logical functions
- Can build basic logic gates
  - **AND:** $w_{ij} = T_j/n$
  - **OR:** $w_{ij} = T_j$
  - **NOT:** use negative weight

$$\text{net}_j = \sum w_{ij}.x_i$$

$$o_j = \frac{1}{1 + \exp\left(-(\text{net}_j - T_j)\right)}$$

- Can build arbitrary logic circuits, finite-state machines and computers given these basis gates.
- Can specify any Boolean function using two layer network (w/ negation)
  - DNF and CNF are universal representations

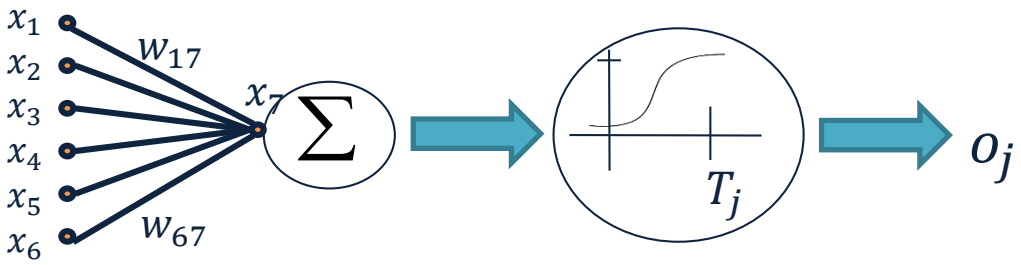
History: Learning Rules

- Hebb (1949) suggested that if two units are both active (firing) then the weights between them should increase: $w_{ij} = w_{ij} + R o_i o_j$
  - R and is a constant called **the learning rate**
  - Supported by physiological evidence
- Rosenblatt (1959) suggested that when a target output value is provided for a single neuron **with fixed input**, it can **incrementally change weights** and learn to produce the output using the **Perceptron learning rule**.
  - assumes **binary output** units; single linear threshold unit
  - Led to the Perceptron Algorithm
- See: http://people.idsia.ch/~juergen/who-invented-backpropagation.html

Examples of update rules

Perceptron Learning Rule

- Given:
  - the **target** output for the output unit is t_j
  - the **input** the neuron sees is x_i
  - the **output** it **produces** is o_j
- Update weights according to $w_{ij} \leftarrow w_{ij} + R\left(t_j - o_j\right)x_i$
  - If output is **correct**, don't change the weights
  - If output is **wrong**, change weights for all inputs which are 1
    - If output is low (0, needs to be 1) increment weights
    - If output is high (1, needs to be 0) decrement weights



Widrow-Hoff Rule

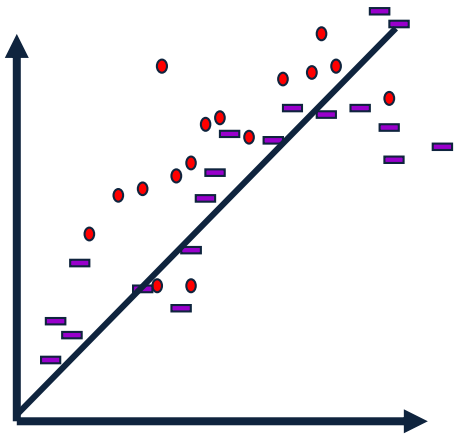
- This incremental update rule provides an approximation to the goal:
  - Find the best linear approximation of the data

$$Err(\vec{w}^{(j)}) = \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

  - where:

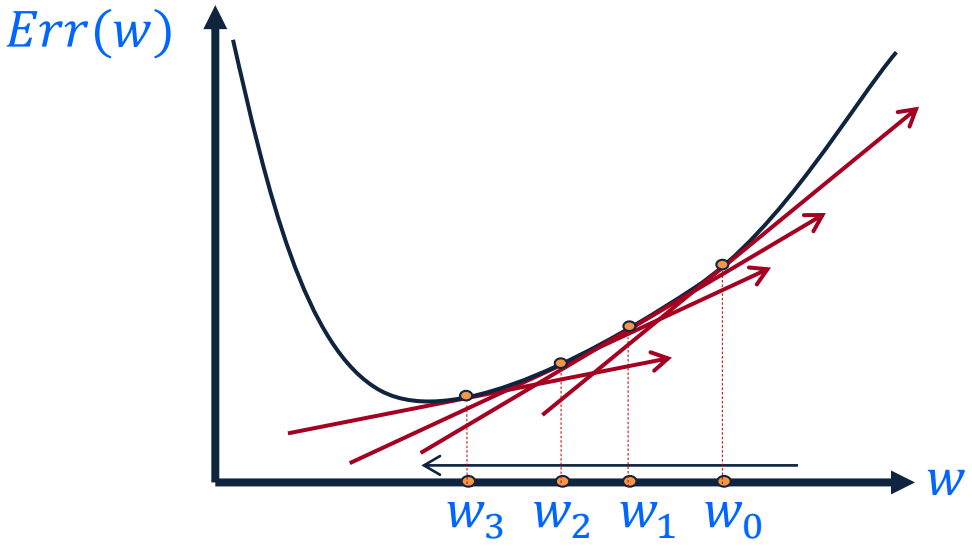
$$o_d = \sum_i w_{ij}.x_i = \vec{w}^{(j)}.\vec{x}$$

    output of linear unit on example *d*
  - t_d = Target output for example d



Gradient Descent

- We use gradient descent determine the weight vector that minimizes $Err(\vec{w}^{(j)})$;
- Fixing the set D of examples, E is a function of $\vec{w}^{(j)}$
- At each step, the weight vector is modified in the direction that produces the steepest descent along the error surface.



Summary: Single Layer Network

- Variety of update rules
  - Multiplicative
  - Additive
- **Batch** and **incremental** algorithms
- Various convergence and efficiency conditions
- There are other ways to learn linear functions
  - Linear Programming (general purpose)
  - Probabilistic Classifiers (some assumption)
- Key algorithms are driven by gradient descent

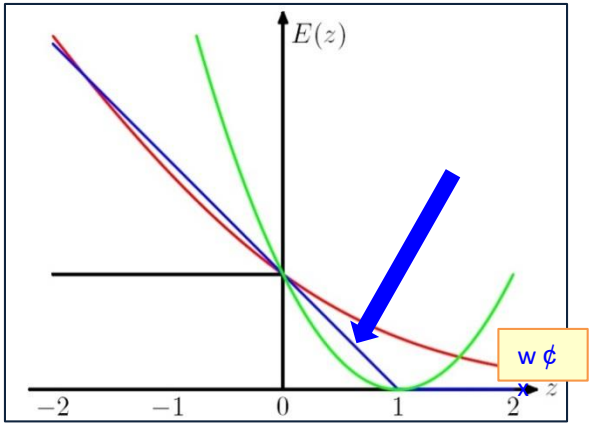
Stochastic Gradient Algorithms

$$w_{t+1} = w_t - r_t\, g_w\, Q(z_t, w_t) = w_t - r_t\, g_t$$

- LMS: $Q((x, y), w) = 1/2\ (y - w \cdot x)^2$
- leads to the update rule (Also called Widrow's Adaline):

$$w_{t+1} = w_t + r\,(y_t - w_t \cdot x_t)\, x_t$$

- Here, even though we make binary predictions based on sign $(w \cdot x)$ we do not take the sign of the dot-product into account in the loss.



- Another common loss function is:
- Hinge loss:

  $Q((x, y), w) = \max(0, 1 - y\, w \cdot x)$
- This leads to the perceptron update rule:

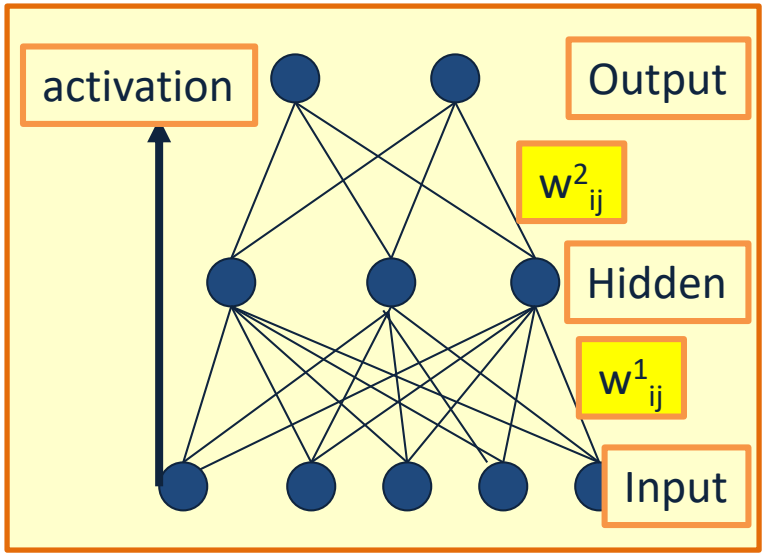
- If $y_i\, w_i \cdot x_i > 1$ (No mistake, by a margin): No update
- Otherwise (Mistake, relative to margin): $w_{t+1} = w_t + r\, y_t\, x_t$

Summary: Single Layer Network

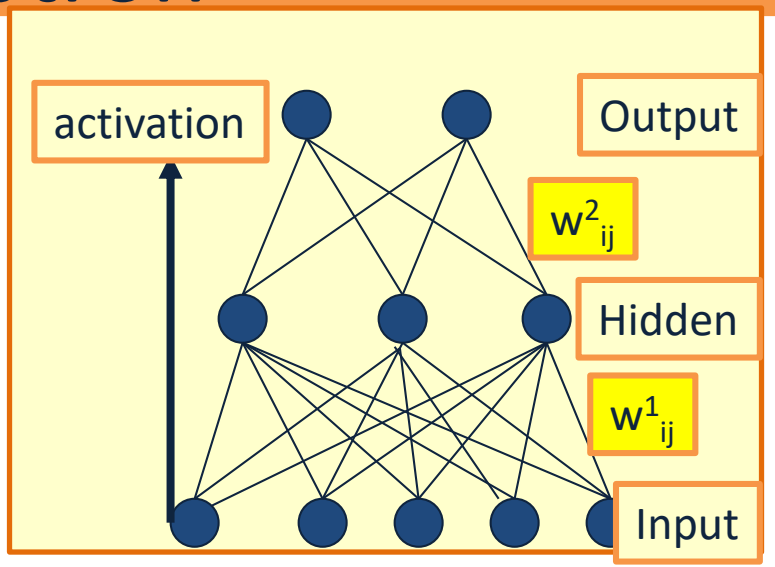
- Variety of update rules
  - Multiplicative
  - Additive
- **Batch** and **incremental** algorithms
- Various convergence and efficiency conditions
- There are other ways to learn linear functions
  - Linear Programming (general purpose)
  - Probabilistic Classifiers (some assumption)
- Key algorithms are driven by gradient descent
- However, the representational restriction is limiting in many applications

Learning with a Multi-Layer Perceptron

- It's easy to learn the top layer – it's just a linear unit.
- Given feedback (truth) at the top layer, and the activation at the layer below it, you can use the Perceptron update rule (more generally, gradient descent) to updated these weights.
- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).



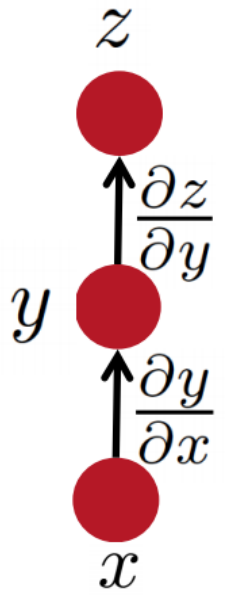
Learning with a Multi-Layer Perceptron



- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).
- Solution: If all the activation functions are differentiable, then the **output** of the network is also a differentiable function of the input and weights in the network.
- Define an error function (e.g., sum of squares) that is a differentiable function of the output, i.e. this error function is also a differentiable function of the weights.
- We can then evaluate the derivatives of the error with respect to the weights, and use these derivatives to find weight values that minimize this error function, using gradient descent (or other optimization methods).
- This results in an algorithm called back-propagation.

Some facts from real analysis

- Simple chain rule
  - If z is a function of y, and y is a function of x
    - Then z is a function of x, as well.
  - Question: how to find $\frac{\partial z}{\partial x}$



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

Reminder

We will use these facts to derive the details of the Backpropagation algorithm.

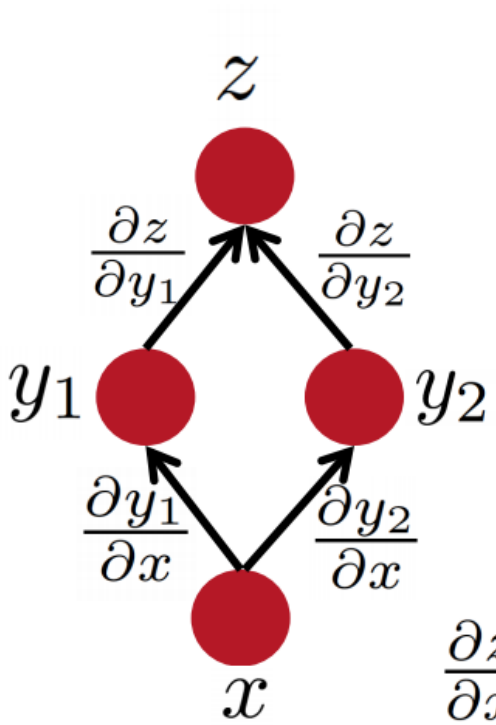
z will be the error (loss) function.
- We need to know how to differentiate z

Intermediate nodes use a logistics function (or another differentiable step function).
- We need to know how to differentiate it.

Slide Credit: Richar

Some facts from real analysis

- Multiple path chain rule



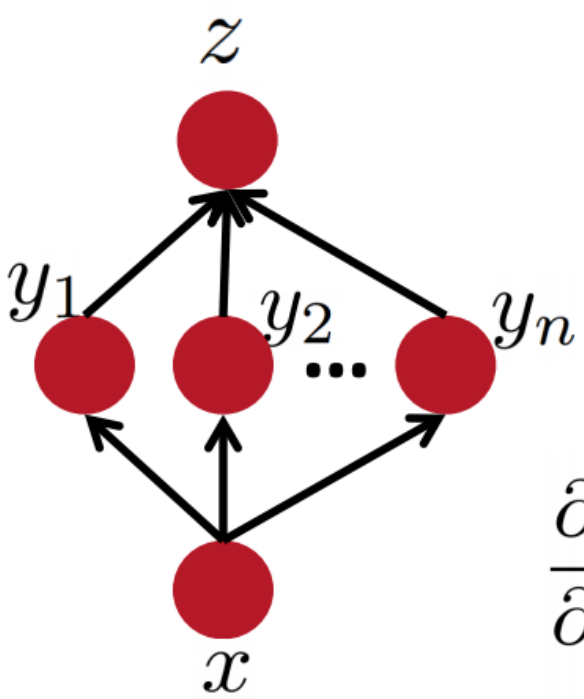
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

Reminder

Slide Credit: Richard Socher

Some facts from real analysis

- Multiple path chain rule: general



$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Reminder

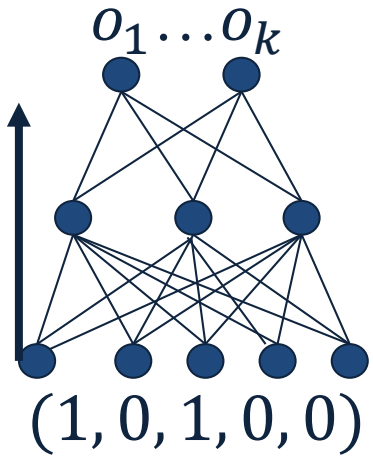
Slide Credit: Richard Socher

Backpropagation Learning Rule

- Since there could be multiple output units, we define the **error** as the sum over all the network output units.

$$Err(\vec{w}) = \frac{1}{2}\sum_{d\in D}\sum_{k\in K}(t_{kd} - o_{kd})^2$$

  - where D is the set of training examples,
  - K is the set of output units



- This is used to derive the (global) learning rule which performs gradient descent in the weight space in an attempt to minimize the error function.

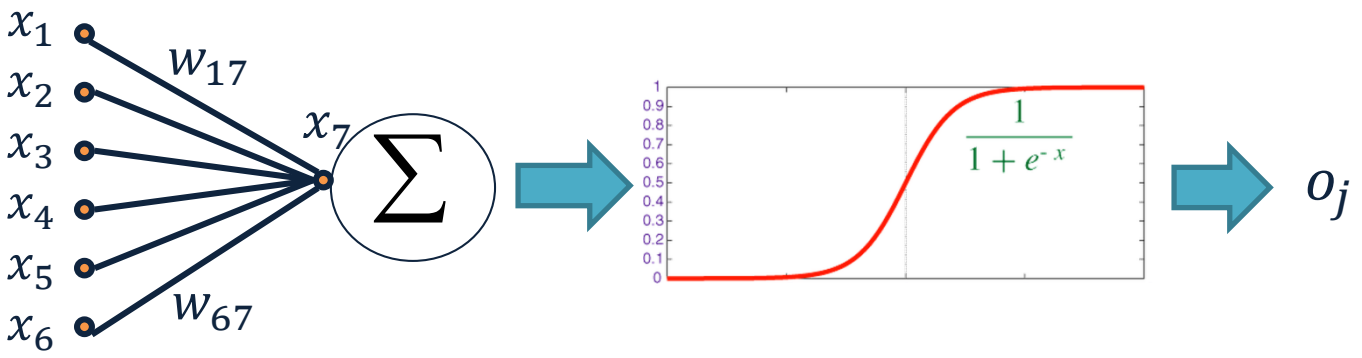
$$\Delta w_{ij} = -R\frac{\partial E}{\partial w_{ij}}$$

Function 1

Reminder: Model Neuron (Logistic)

Neuron Definition

- Neuron is modeled by a unit j connected by weighted links w_{ij} to other units i.

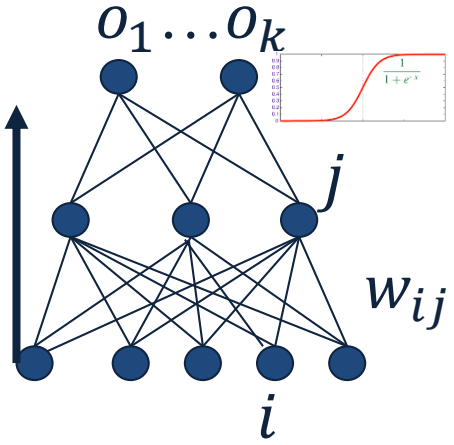


  - Use a non-linear, differentiable output function such as the sigmoid or logistic function
  - Net input to a unit is defined as: $\text{net}_j = \sum w_{ij}.x_i$ — Function 2
  - Output of a unit is defined as: $o_j = \frac{1}{1+\exp\left(-(\text{net}_j - T_j)\right)}$ — Function 3

Derivatives

- Function 1 (error):
  - $y = \frac{1}{2}\sum_{k \in K}(c_k - x_k)^2$
  - $\frac{\partial y}{\partial x_i} = -(t_i - x_i)$
- Function 2 (linear gate):
  - $y = \sum w_i . x_i$
  - $\frac{\partial y}{\partial w_i} = x_i$
- Function 3 (differentiable step function):
  - $y = \frac{1}{1+\exp\{-(x-T)\}}$
  - $\frac{\partial y}{\partial x} = \frac{\exp\{-(x-T)\}}{(1+\exp\{-(x-T)\})^2} = y(1-y)$

Propagation error to earlier layer



Derivation of Learning Rule

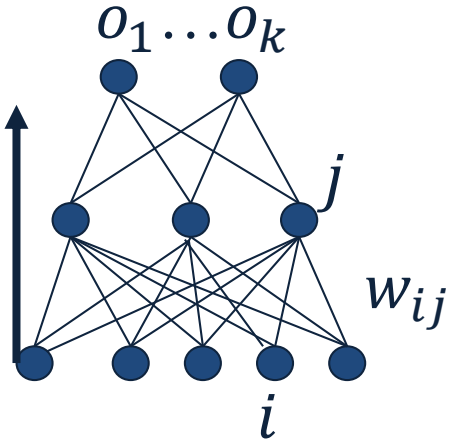
- The weights are updated incrementally; the error is computed **for each example** and the weight update is then derived.
- $Err_d(\vec{w}) = \frac{1}{2}\sum_{k \in K}(t_k - o_k)^2$
- w_{ij} influences the output only through net_j

$$\text{net}_j = \sum w_{ij}.x_{ij}$$

- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}}$$

Propagation error to earlier layer



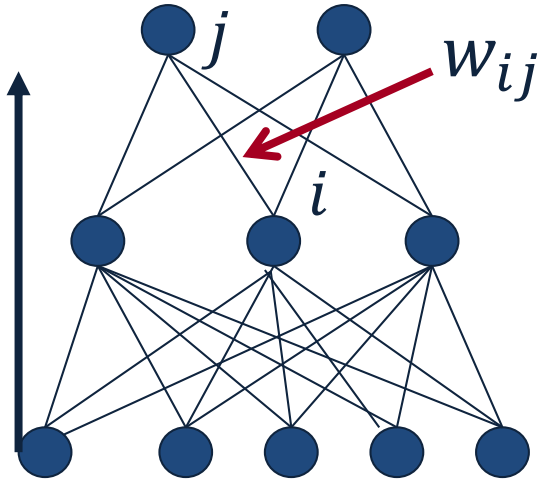
Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through net_j
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

$$= \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

$$= -(t_j - o_j)\, o_j(1 - o_j)\, x_{ij}$$



$$Err_d(\vec{w}) = \frac{1}{2}\sum_{k \in K}(t_k - o_k)^2$$

$$\frac{\partial o_j}{\partial \text{net}_j} = o_j(1 - o_j)$$

$$o_j = \frac{1}{1 + \exp\{-(\text{net}_j - T_j)\}}$$

$\sum w_{ij}.x_{ij}$

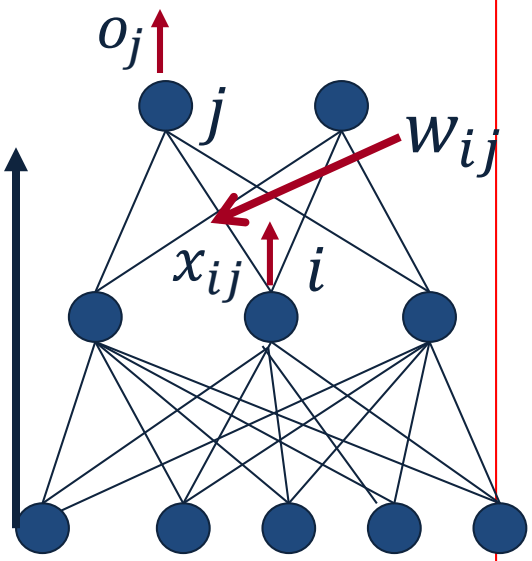
Derivation of Learning Rule (3)

- Weights of output units:
  - w_{ij} is changed by:

$$\Delta w_{ij} = R(t_j - o_j)o_j(1 - o_j)x_{ij}$$
$$= R\delta_j x_{ij}$$

where

$$\delta_j = (t_j - o_j)o_j(1 - o_j)$$



Derivation of Learning Rule (4)

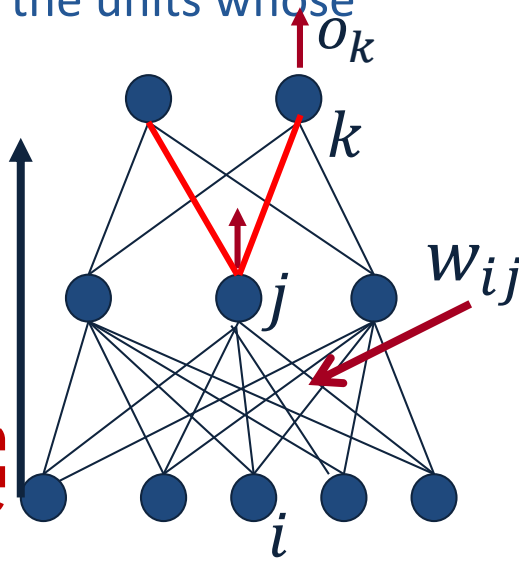
- Weights of hidden units:
  - w_{ij} Influences the output only through all the units whose direct input include j

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} =$$

$$\text{net}_j = \sum w_{ij}.x_{ij}$$

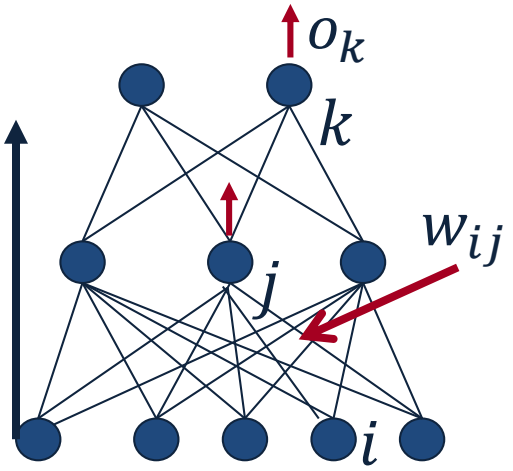
$$= \sum_{k \in downstream(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} x_{ij}$$

$$= \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_{ij}$$



Derivation of Learning Rule (5)

- Weights of hidden units:
  - w_{ij} influences the output only through all the units whose direct input include j



$$\frac{\partial E_d}{\partial w_{ij}} = \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_{ij} =$$

$$= \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} x_{ij}$$

$$= \sum_{k \in downstream(j)} -\delta_k \, w_{jk} \, o_j(1 - o_j) \, x_{ij}$$

Derivation of Learning Rule (6)

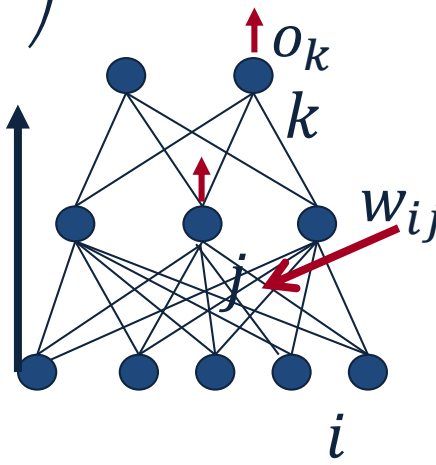
- Weights of hidden units:
  - w_{ij} is changed by:

$$\Delta w_{ij} = R\, o_j(1-o_j).\left(\sum_{k \in downstream(j)} -\delta_k\ w_{jk}\right) x_{ij}$$

$$= R\delta_j x_{ij}$$

- Where

$$\delta_j = o_j(1-o_j).\left(\sum_{k \in downstream(j)} -\delta_k\ w_{jk}\right)$$



Delta Rule

- First determine the error for the output units.
- Then, backpropagate this error layer by layer through the network, changing weights appropriately in each layer.

The Backpropagation Algorithm

- Create a fully connected three layer network. Initialize weights.
- Until all examples produce the correct output within ϵ (or other criteria)

For each example in the training set do:

1. Compute the network output for this example
2. Compute the error between the output and target value

$$\delta_k = (t_k - o_k)o_k(1 - o_k)$$

1. For each output unit k, compute error term

$$\delta_j = o_j(1 - o_j). \sum_{k \in downstream(j)} -\delta_k \; w_{jk}$$

1. For each hidden unit, compute error term:

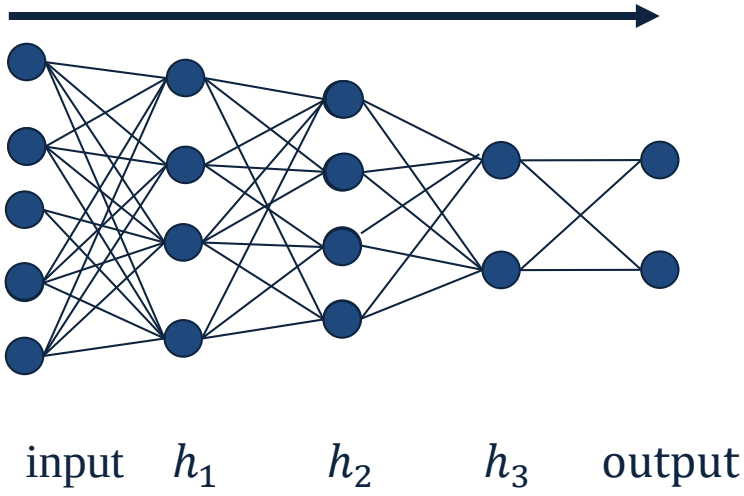
$$\Delta w_{ij} = R\delta_j x_{ij}$$

1. Update network weights

End epoch

More Hidden Layers

- The same algorithm holds for more hidden layers.



Comments on Training

- **No guarantee of convergence**; may **oscillate** or reach a local minima.
- In practice, many large networks can be trained on **large amounts of data** for realistic problems.
- **Many epochs** (tens of thousands) may be needed for adequate training. Large data sets may require many hours of CPU
- **Termination criteria**: Number of epochs; Threshold on training set error; No decrease in error; Increased error on a validation set.
- To **avoid local minima**: several trials with different random initial weights with majority or voting techniques

Over-training Prevention

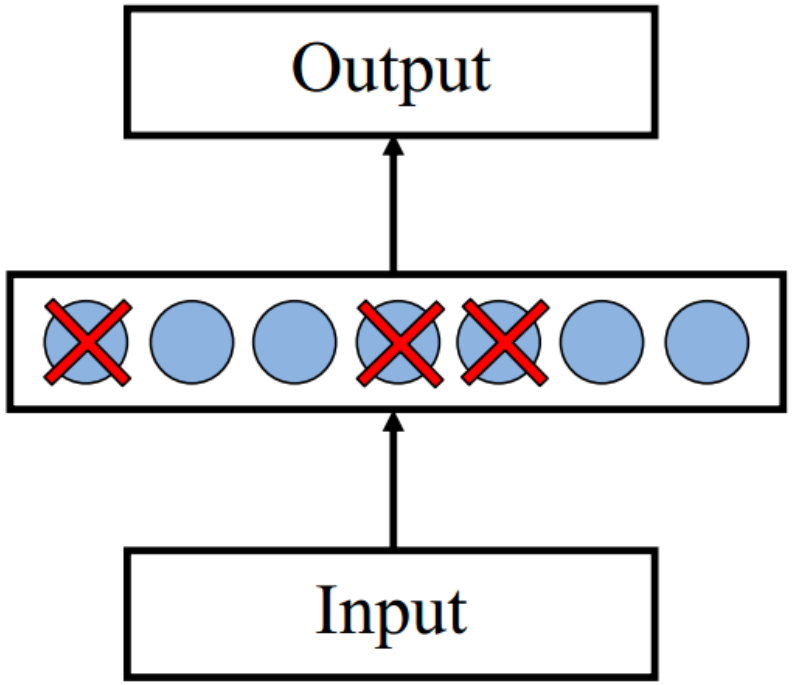
- Running too many epochs may **over-train** the network and result in over-fitting. (improved result on training, decrease in performance on test set)
- Keep an **hold-out validation** set and test accuracy after every epoch
- Maintain weights for best performing network on the validation set and return it when performance decreases significantly beyond that.
- To avoid losing training data to validation:
  - Use 10-fold cross-validation to determine the average number of epochs that optimizes validation performance
  - Train on the full data set using this many epochs to produce the final results

Over-fitting prevention

- **Too few hidden units** prevent the system from adequately fitting the data and learning the concept.
- **Using too many hidden units** leads to over-fitting.
- Similar cross-validation method can be used to determine an appropriate number of hidden units. (general)
- Another approach to prevent over-fitting is weight-decay: all weights are multiplied by some fraction in (0,1) after every epoch.
  - Encourages smaller weights and less complex hypothesis
  - Equivalently: change Error function to include a term for the sum of the squares of the weights in the network. (general)

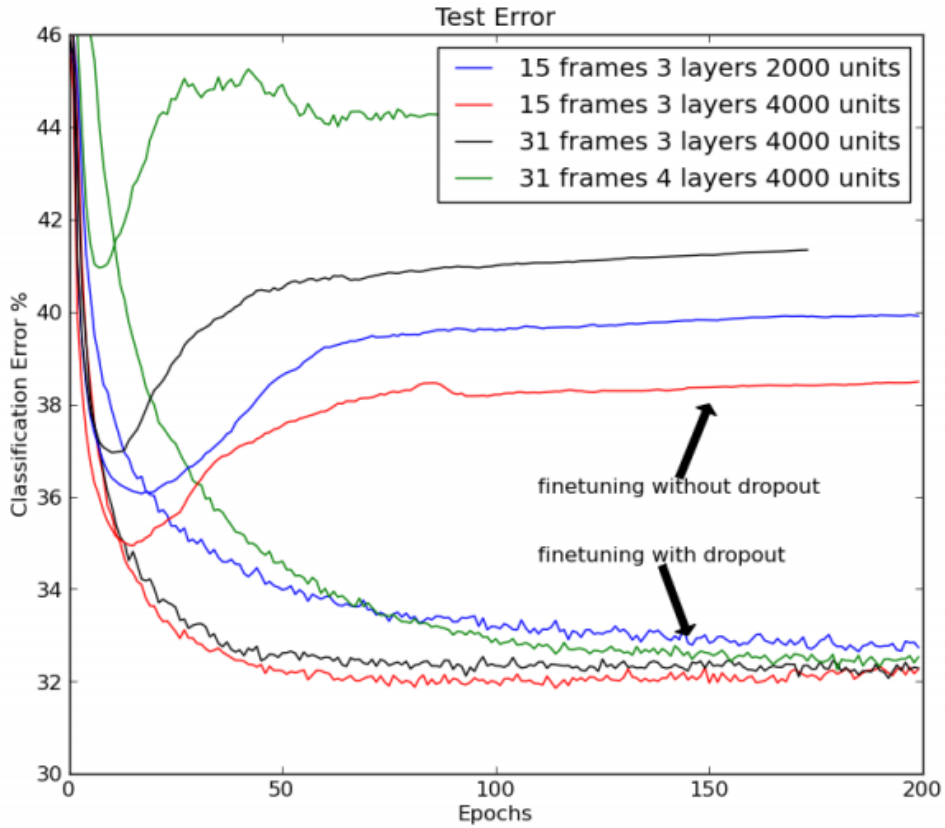
Dropout training

- Proposed by (Hinton et al, 2012)



- Each time decide whether to delete one hidden unit with some probability *p*

Dropout training

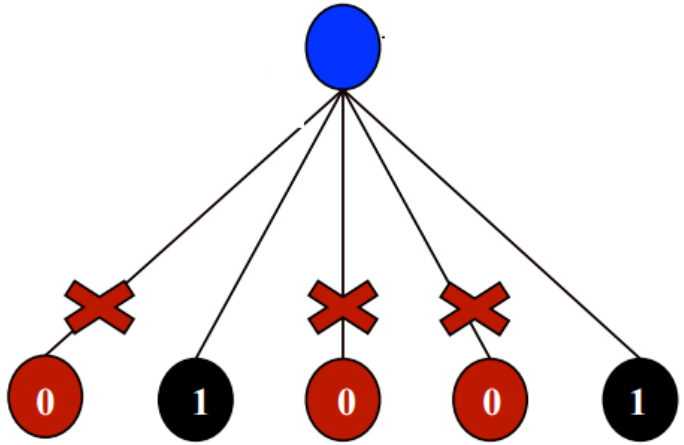


- Dropout of 50% of the hidden units and 20% of the input units (Hinton et al, 2012)

Dropout training

- Model averaging effect
  - Among 2^H models, with shared parameters
    - H: number of units in the network
  - Only a few get trained
  - Much stronger than the known regularizer

- What about the input space?
  - Do the same thing!



Input-Output Coding

One way to do it, if you start with a collection of sparsely representation examples, is to use dimensionality reduction methods:

- Your m examples are represented as a m x 10^6 matrix
- Multiple it by a random matrix of size 10^6 x 300, say.
- Random matrix: Normal(0,1)
- New representation: m x 300 dense rows

...uts can make ...generalization.

...rate input unit;

- For multi-valued features include one binary unit per value rather than trying to encode input information in fewer units.
  - Very common today to use distributed representation of the input – real valued, dense representation.
- For disjoint categorization problem, best to have one output unit for each category rather than encoding N categories into log N bits.

Representational Power

- The Backpropagation version presented is for networks with a single hidden layer,

But:

- Any Boolean function can be represented by a **two layer** network (simulate a two layer AND-OR network)
- Any **bounded continuous function** can be approximated with **arbitrary small error** by a **two layer** network.
- Sigmoid functions provide a set of **basis function** from which arbitrary function can be composed.
- **Any function** can be approximated to arbitrary accuracy by a **three layer** network.

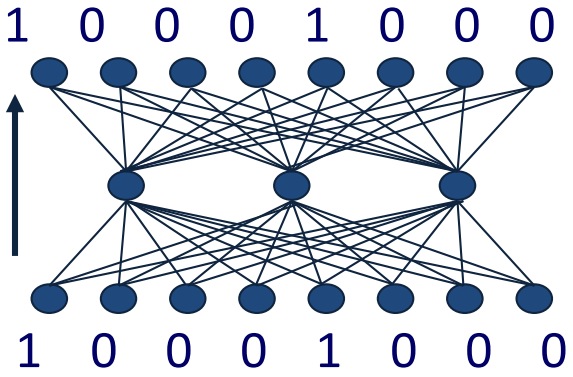
Hidden Layer Representation

- Weight tuning procedure sets weights that define whatever hidden units representation is most effective at minimizing the error.
- Sometimes Backpropagation will define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.
- Trained hidden units can be seen as newly constructed features that re-represent the examples so that they are linearly separable

Auto-associative Network

- An auto-associative network trained with 8 inputs, 3 hidden units and 8 output nodes, where the output must reproduce the input.
- When trained with vectors with only one bit on

INPUT	HIDDEN		
1 0 0 0 0 0 0 0	.89	.40	0.8
0 1 0 0 0 0 0 0	.97	.99	.71
….			
0 0 0 0 0 0 0 1	.01	.11	.88



- Learned the standard 3-bit encoding for the 8 bit vectors.
- Illustrates also data compression aspects of learning

Sparse Auto-encoder

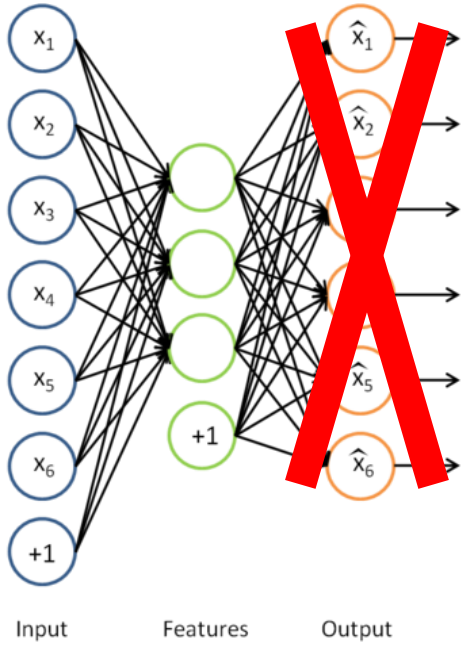
- Encoding: $\boldsymbol{y} = f(W\boldsymbol{x} + \boldsymbol{b})$
- Decoding: $\hat{\boldsymbol{x}} = g(W'\boldsymbol{y} + \boldsymbol{b}')$
  - Goal: perfect reconstruction of input vector $\boldsymbol{x}$, by the output $\hat{\boldsymbol{x}} = h_{\boldsymbol{\theta}}(\boldsymbol{x})$
    - Where $\boldsymbol{\theta} = \{\boldsymbol{W}, \boldsymbol{W}'\}$
  - Minimize an error function $\boldsymbol{l}(h_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{x})$
    - For example:

$$l(h_\theta(\boldsymbol{x}), \boldsymbol{x}) = \|h_\theta(\boldsymbol{x}) - \boldsymbol{x}\|^2$$

  - And regularize it

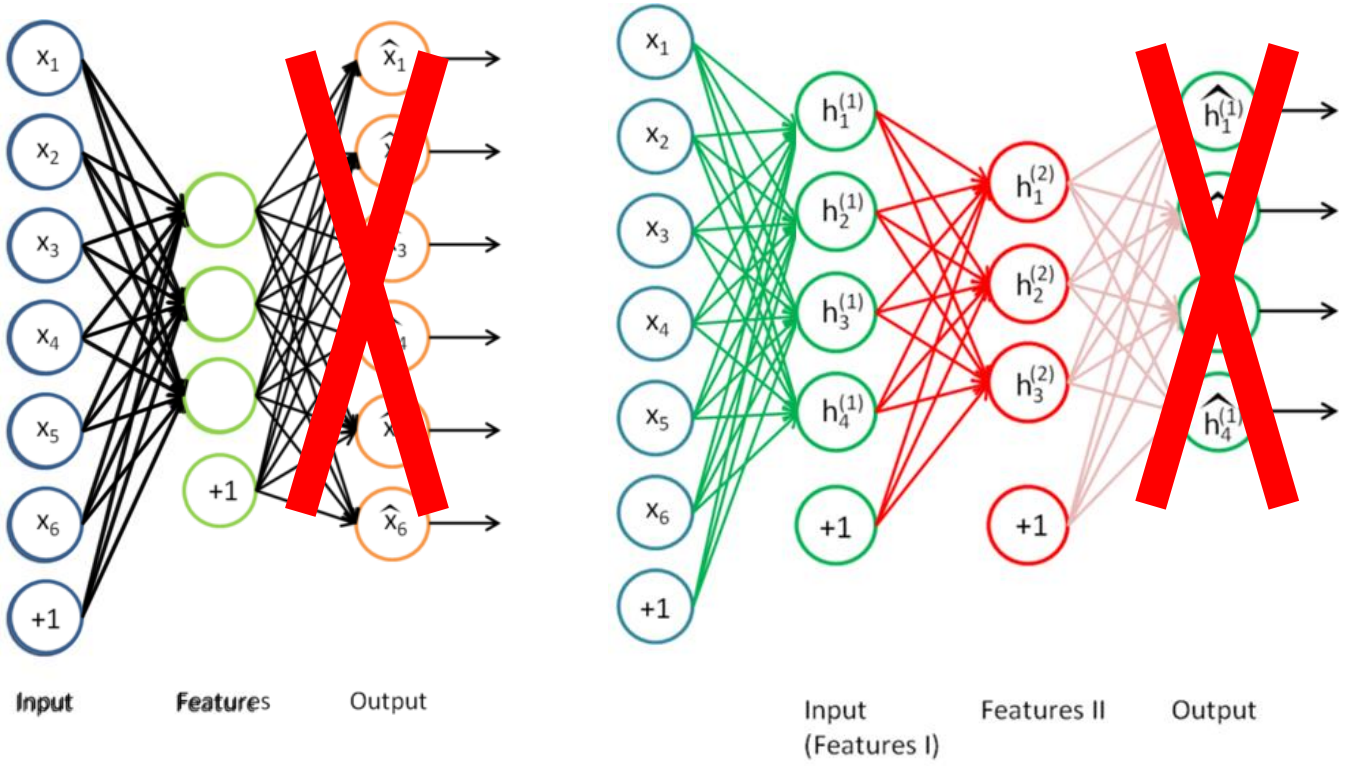
$$\min_\theta \sum_{\boldsymbol{x}} l\left(h_\theta(\boldsymbol{x}), \boldsymbol{x}\right) + \sum_i |w_i|$$

- After optimization drop the reconstruction layer and add a new layer



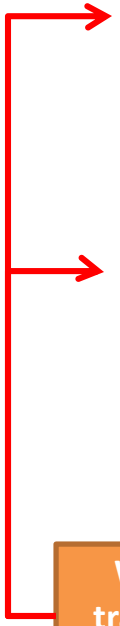
Stacking Auto-encoder

- Add a new layer, and a reconstruction layer for it.
- And try to tune its parameters such that
- And continue this for each layer



Beyond supervised learning

- So far what we had was purely **supervised**.
  - Initialize parameters randomly
  - Train in supervised mode typically, using backprop
  - Used in most practical systems (e.g. speech and image recognition)
- Unsupervised, layer-wise + supervised classifier on top
  - Train each layer unsupervised, one after the other
  - Train a supervised classifier on top, keeping the other layers fixed
  - Good when very few labeled samples are available
- Unsupervised, layer-wise + global supervised fine-tuning
  - Train each layer unsupervised, one after the other
  - Add a classifier layer, and retrain the whole thing supervised
  - Good when label set is poor (e.g. pedestrian detection)



We won't talk about unsupervised pre-training here. But it's good to have this in mind, since it is an active topic of research.