COMP4901K/Math4824B
Machine Learning for Natural Language Processing

Lecture 10: Neural Networks

Instructor: Yangqiu Song

Slides credits: John Canny, Dan Roth

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, and NLP problems

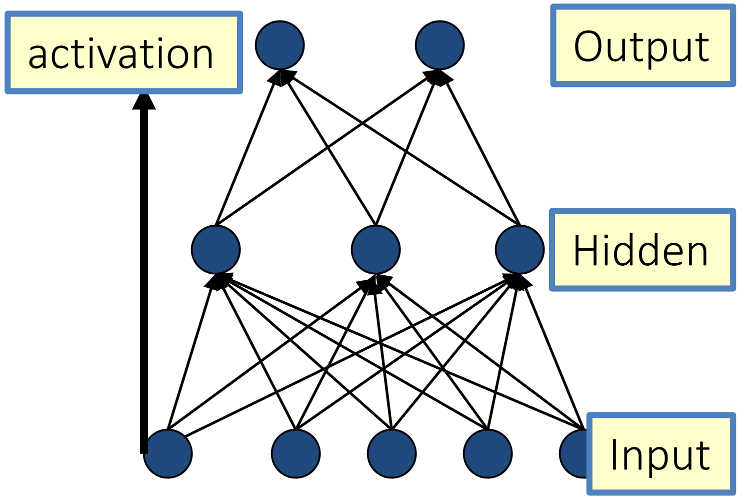
Neural Networks

- Neural Networks are **functions**: $\mathrm{NN}: X \to Y$
  - where $X = [0,1]^n$*, or* $\{0,1\}^n$ *and* $Y = [0,1]$, $\{0,1\}$ (or $\{-1,1\}$)

- 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.

Motivation for Neural Networks

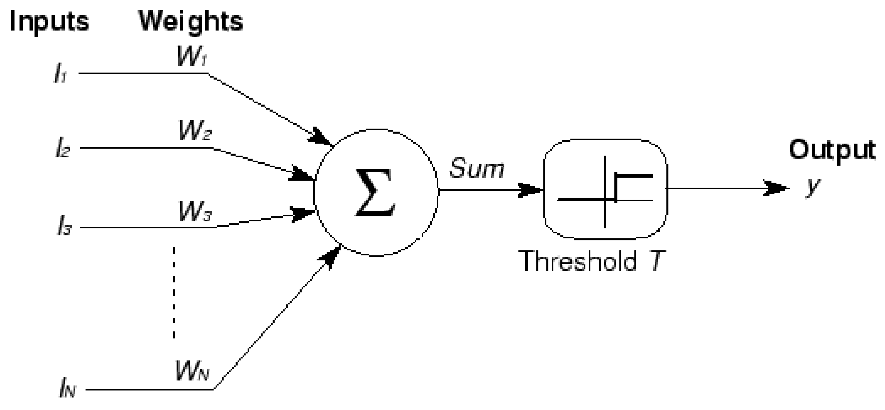
- Many algorithms/principles have been developed long time ago

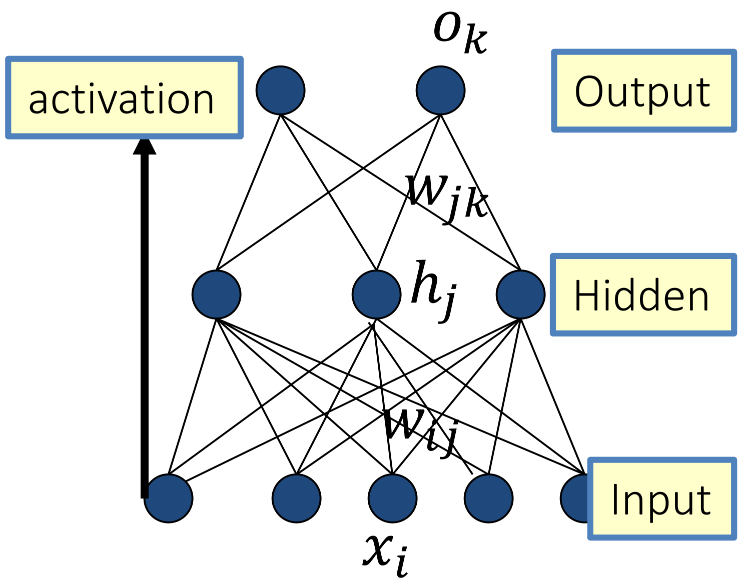
- One potentially interesting perspective:
  - Before we looked at NN only as function approximators.
  - Now, we look at the intermediate representations generated while learning as meaningful
  - Ideas are being developed on the value of these intermediate representations for transfer learning etc.

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

Basic Unit in Multi-Layer Neural Network

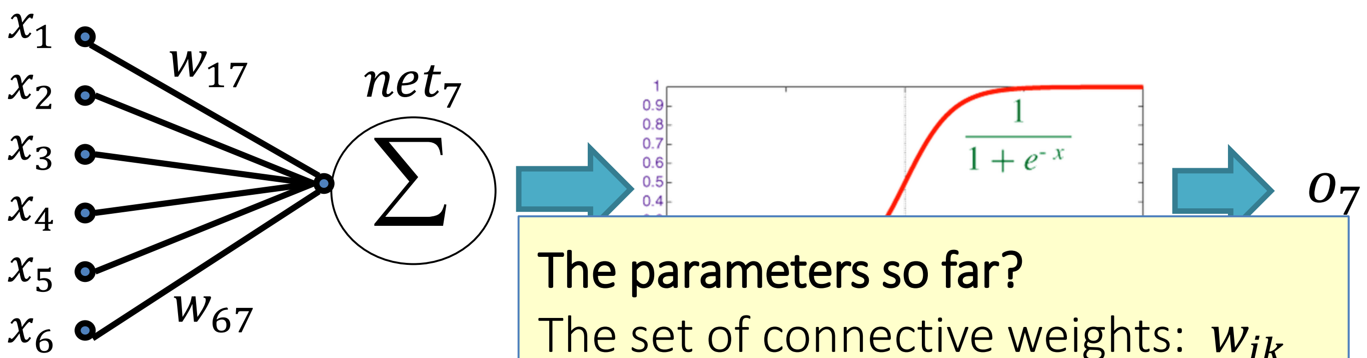
- **Linear Unit**: $o_j = \vec{w}.\vec{x}$ multiple layers of linear functions produce linear functions. We want to represent nonlinear functions.
  - Note: Here we use a slightly different notation for dot product $\vec{w}.\vec{x}$
- **Threshold units:** $o_j = sgn(\vec{w}.\vec{x} - b)$ are not differentiable, hence unsuitable for gradient descent





Model Neuron (Logistic, slightly different notations)

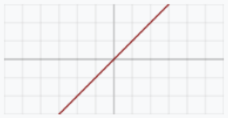
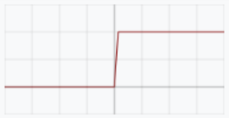
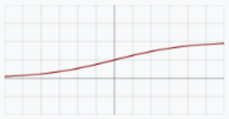
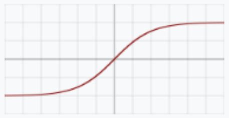
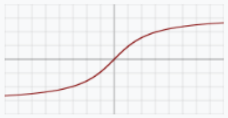
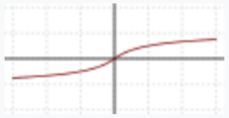
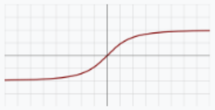
- Neuron is modeled by a unit k connected by weighted links w_{jk} to other units j.
  - Note: we use a different kind of index of w to indicate different neurons



The parameters so far?
The set of connective weights: w_{jk}
The threshold/bias value: b_k

  - Use a non-linear, differentiable output function such as the sigmoid or logistic function
  - Net input to a unit is defined as: $\text{net}_k = \sum w_{jk} . x_j$
  - Output of a unit is defined as: $o_k = \dfrac{1}{1+\exp(-(\text{net}_k - b_k))}$

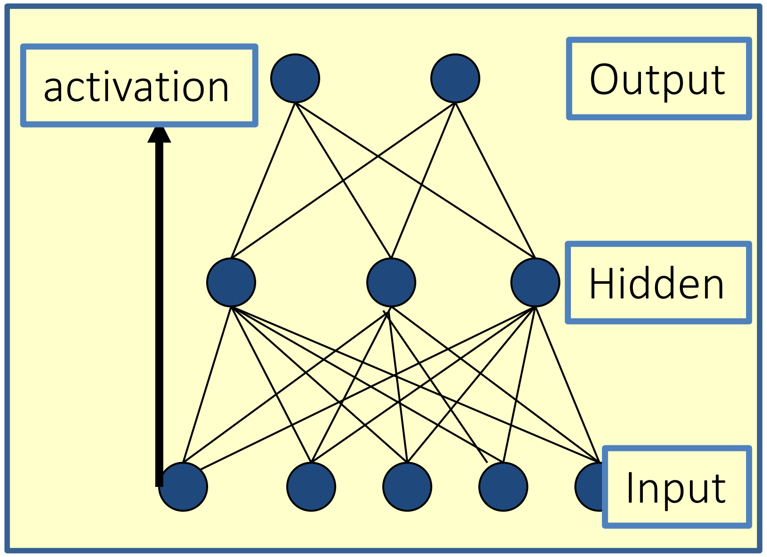
Activation function

Name	Plot	Equation	Derivative (with respect to *x*)	Range
Identity		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$	$\{0, 1\}$
Logistic (a.k.a. Sigmoid or Soft step)		$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$[1]	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
TanH		$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$	$\left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$
Softsign[9][10]		$f(x) = \frac{x}{1 + \|x\|}$	$f'(x) = \frac{1}{(1 + \|x\|)^2}$	$(-1, 1)$
Inverse square root unit (ISRU)[11]		$f(x) = \frac{x}{\sqrt{1 + \alpha x^2}}$	$f'(x) = \left(\frac{1}{\sqrt{1 + \alpha x^2}}\right)^3$	$\left(-\frac{1}{\sqrt{\alpha}}, \frac{1}{\sqrt{\alpha}}\right)$

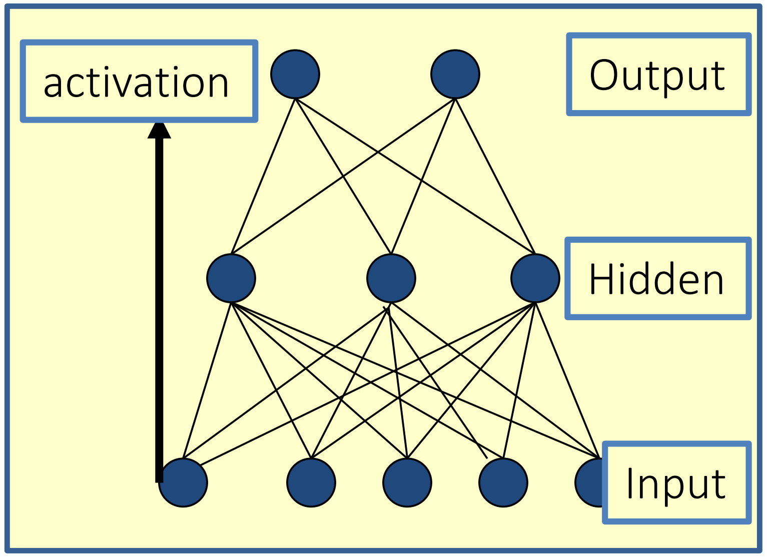
- https://en.wikipedia.org/wiki/Activation_function

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).



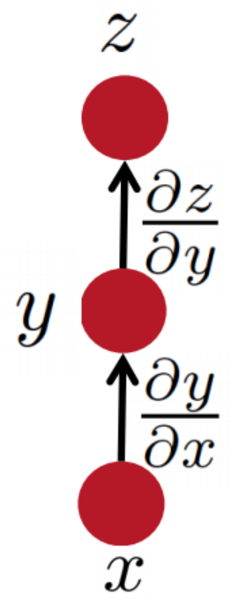
Learning with a Multi-Layer Perceptron



- 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 $\frac{\partial z}{\partial x}$


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

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

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

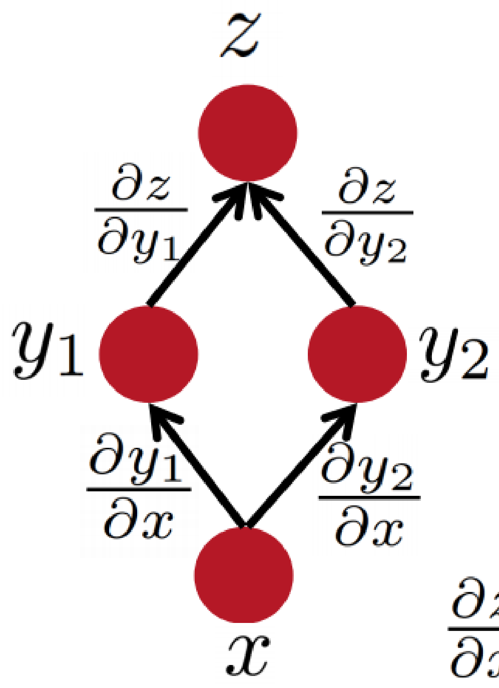
z will be the error (loss) function.
- We need to know how to differentiate z

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

Slide Credit: Richard Socher

Some facts from real analysis

- Multiple path chain rule



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

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

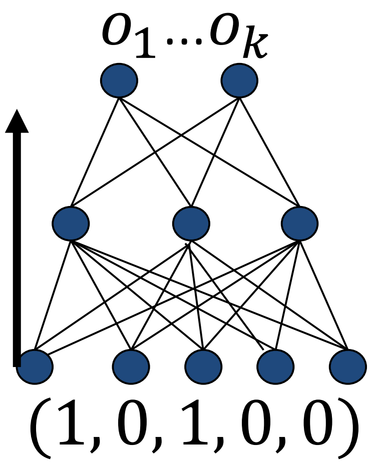
Slide Credit: Richard Socher

Backpropagation Learning Rule

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

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

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



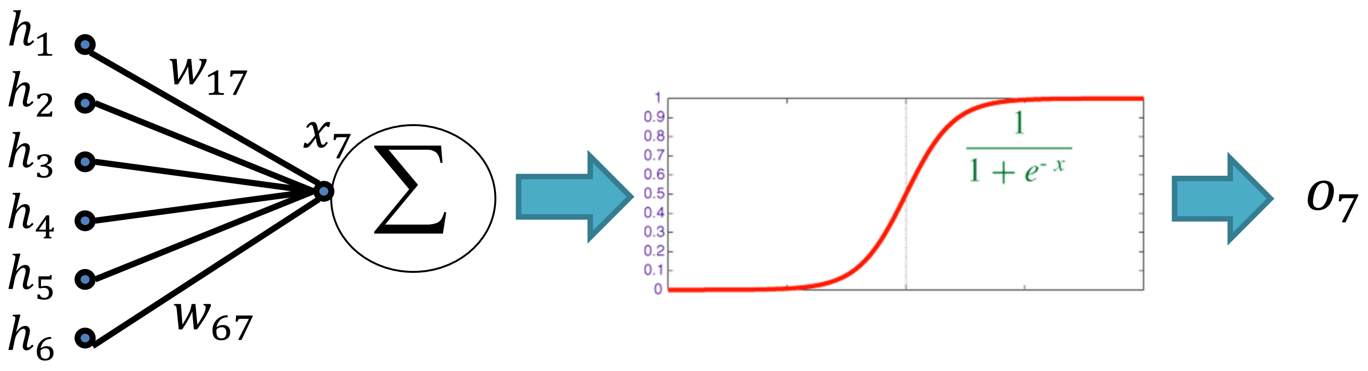
- This is used to derive the (global) learning rule which performs gradient descent in the weight space in an attempt to minimize the error function.

$$\Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}}$$

Function 1

Reminder: Model Neuron (Logistic)

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



  - Use a non-linear, differentiable output function such as the sigmoid or logistic function
  - Net input to a unit is defined as: $\text{net}_k = \sum w_{jk} . h_j$ Function 3
  - Output of a unit is defined as: $o_k = \dfrac{1}{1 + \exp(-(\text{net}_k - b_k))}$ Function 2

Derivation of Learning Rule

- The weights are updated incrementally (SGD)
  - the error is computed **for each example** and the weight update is then derived.

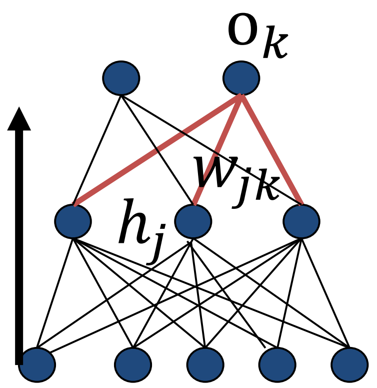
$$Err_d(\vec{w}) = \frac{1}{2}\sum_{k\in K}(t_k - o_k)^2$$

- w_{jk} influences the output only through net_k

$$\mathrm{net}_k = \sum w_{jk}.h_j$$

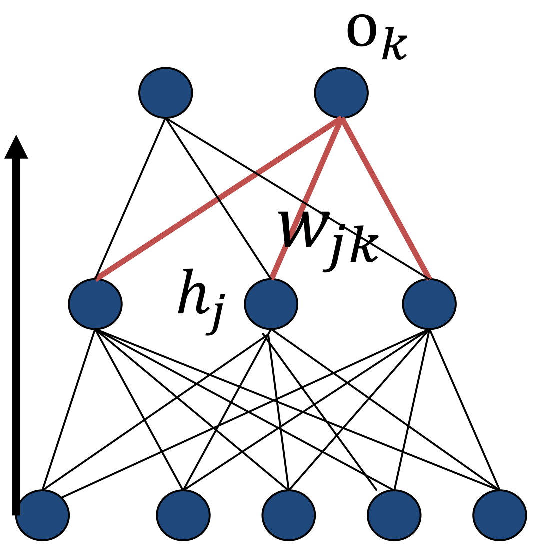
- Therefore:

$$\frac{\partial E_d}{\partial w_{jk}} = \frac{\partial E_d}{\partial \mathrm{net}_k}\frac{\partial \mathrm{net}_k}{\partial w_{jk}}$$



Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{jk} influences the output only through $\mathbf{net}_k$
- Therefore:



$$\frac{\partial E_d}{\partial w_{jk}} = \frac{\partial E_d}{\partial \mathrm{net}_k}\frac{\partial \mathrm{net}_k}{\partial w_{jk}}$$

$$= \boxed{\frac{\partial E_d}{\partial \mathrm{o}_k}}\boxed{\frac{\partial o_k}{\partial \mathrm{net}_k}}\boxed{\frac{\partial \mathrm{net}_k}{\partial w_{jk}}}$$

$$= \boxed{-(t_k - o_k)}\boxed{o_k(1-o_k)}\boxed{h_j}$$

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

$$\frac{\partial o_k}{\partial \mathrm{net}_k} = o_k(1-o_k)$$

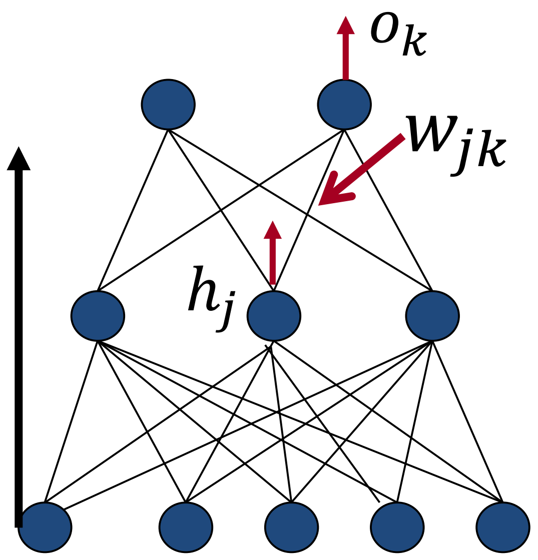
$$o_k = \frac{1}{1+\exp\{-(\mathrm{net}_k - b_k)\}}$$

$$\mathrm{net}_k = \sum w_{jk}.h_j$$

Derivation of Learning Rule (3)

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

$$\Delta w_{jk} = \alpha(t_k - o_k)o_k(1 - o_k)h_j$$
$$= \alpha\delta_k h_j$$



where we set

$$\delta_k = \frac{\partial E_d}{\partial \mathrm{net}_k} = \frac{\partial E_d}{\partial o_k}\frac{\partial o_k}{\partial \mathrm{net}_k} = (t_k - o_k)o_k(1 - o_k)$$

- α is the learning rate

Derivation of Learning Rule (4)

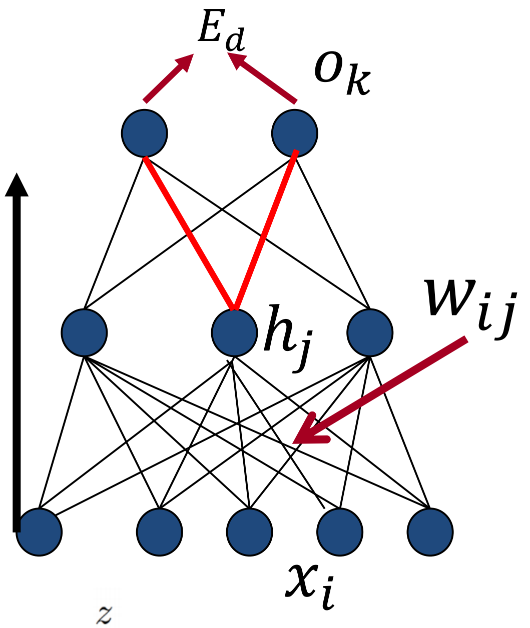
- Weights of hidden units:
  - w_{ij} Influences the output only through all the units whose direct input include *j*

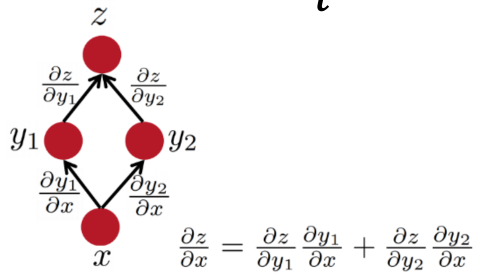
$$\text{net}_j = \sum w_{ij}.x_i$$

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

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

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




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

Derivation of Learning Rule (5)

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

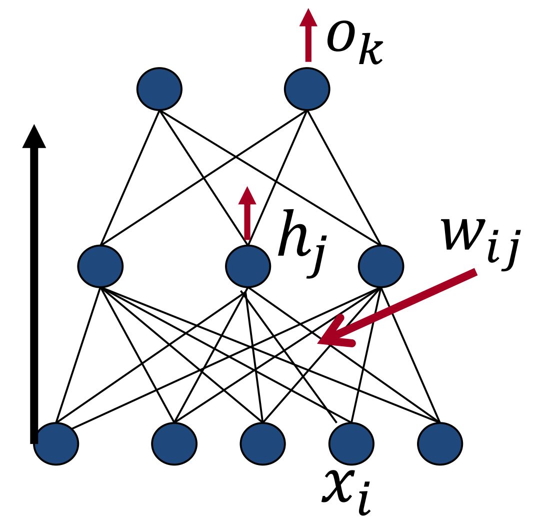
$$\frac{\partial E_d}{\partial w_{ij}} = \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i =$$

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

$$= \sum_{k \in downstream(j)} -\delta_k \boxed{w_{jk}} \boxed{h_j(1-h_j)} \quad x_i$$

$$\boxed{\text{net}_j = \sum w_{jk}. h_j}$$

$$\boxed{\begin{array}{c} \dfrac{\partial h_j}{\partial \text{net}_j} = h_j(1-h_j) \\ h_j = \dfrac{1}{1+\exp\{-(\text{net}_j - b_j)\}} \end{array}}$$



Derivation of Learning Rule (6)

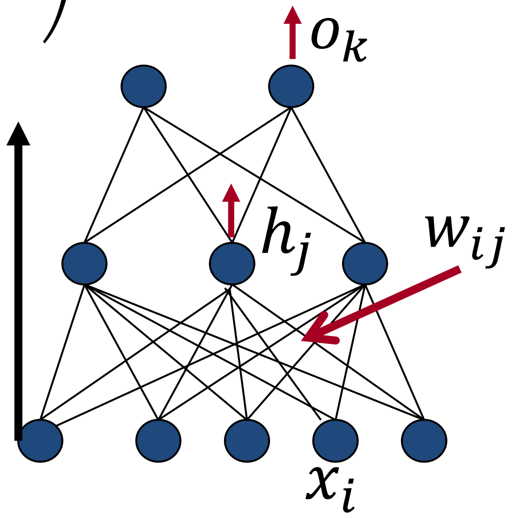
- Weights of hidden units:
  - w_{ij} is changed by:

$$\Delta w_{ij} = \alpha\, h_j(1 - h_j).\left(\sum_{k \in downstream(j)} -\delta_k \; w_{jk}\right) x_i$$

$$= \alpha \delta_j x_i$$

- Where

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



- 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:

- Compute the network output for this example
- Compute the error between the output and target value

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

- For each output unit k, compute error term

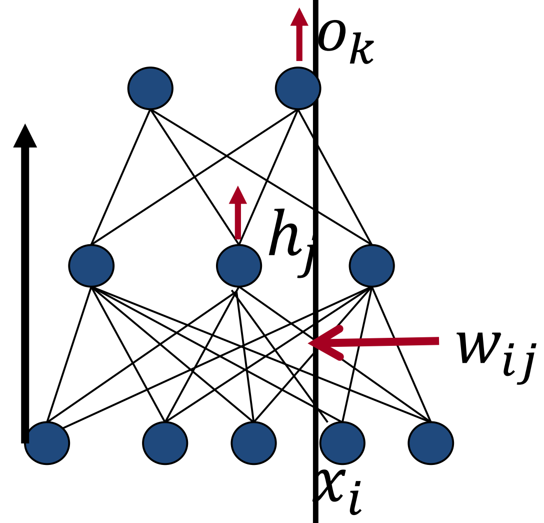
$$\delta_j = h_j(1 - h_j).\sum_{k \in downstream(j)} -\delta_k \ w_{jk}$$

- For each hidden unit, compute error term:

$$\Delta w_{ij} = \alpha \delta_j x_i$$

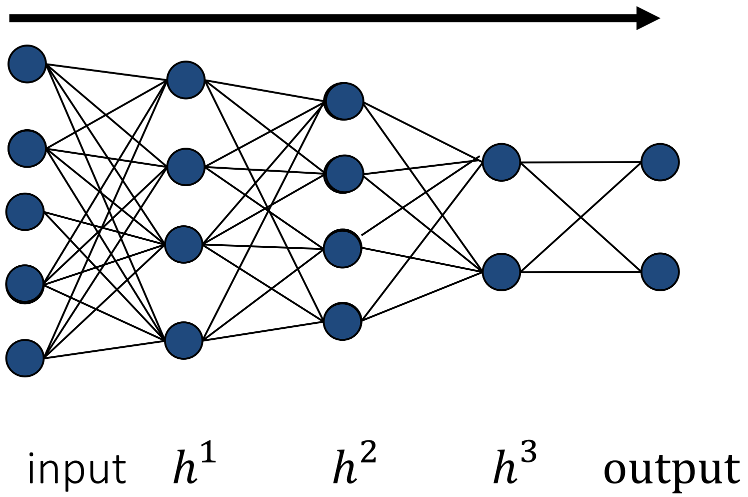
- Update network weights

End epoch



More Hidden Layers

- The same algorithm holds for more hidden layers.



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 $m \times 10^6$ matrix
- Multiple it by a random matrix of size $10^6 \times 300$, say.
- Random matrix: Normal(0,1)
- New representation: $m \times 300$ dense rows

- 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.

Comments on Training

- **No guarantee of convergence**; may 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

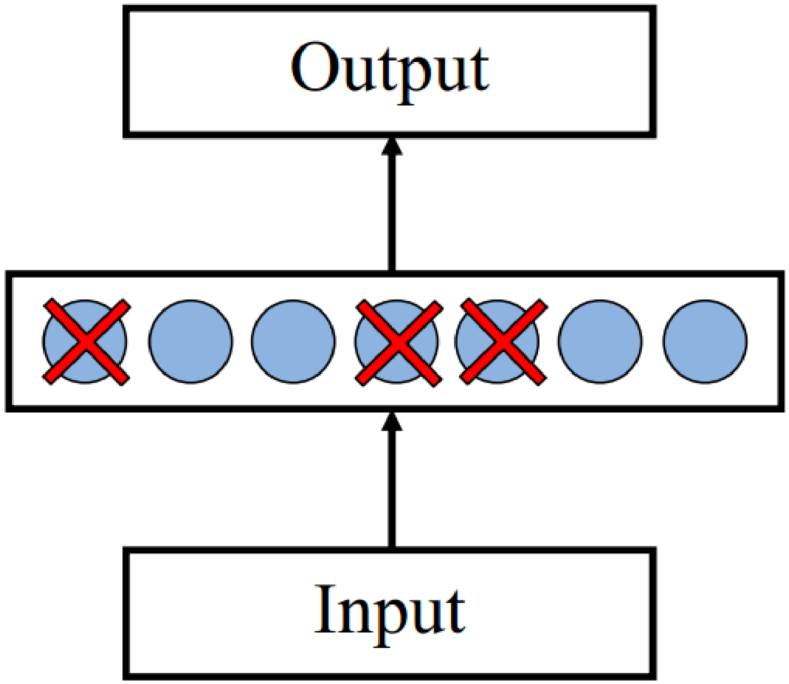
- Running too many epochs may over-train the network and result in over-fitting. (improved result on training, decrease in performance on test set)
- Keep an hold-out validation set and test accuracy after every epoch
- Maintain weights for best performing network on the validation set and return it 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. (L2 norm regularization)

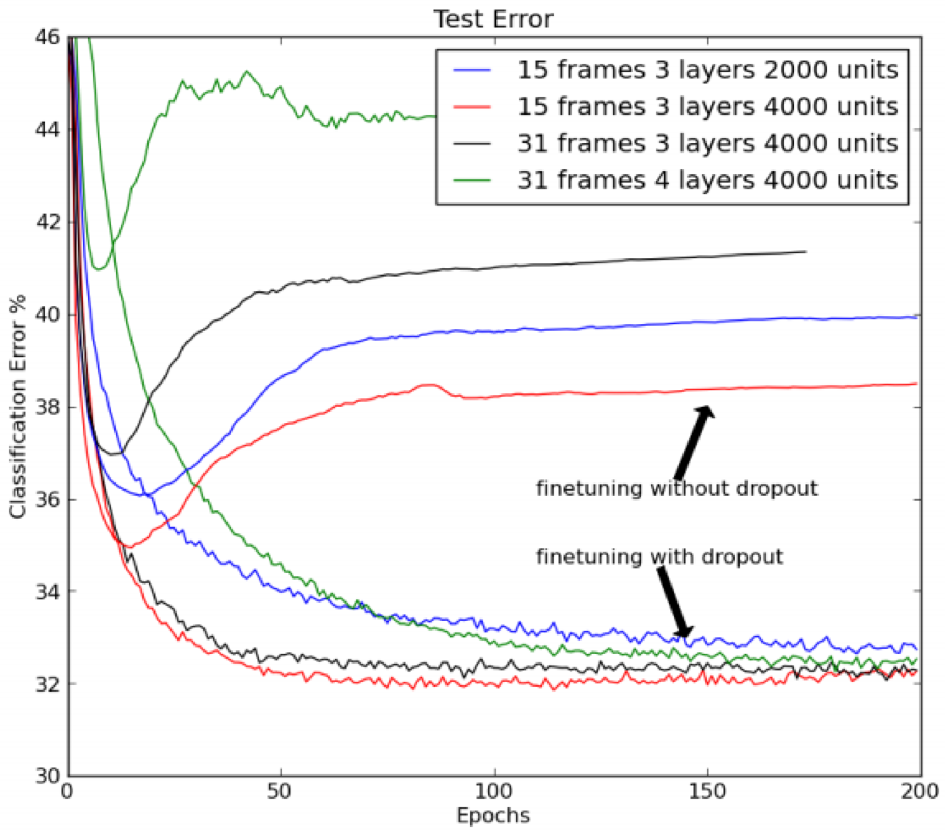
Dropout training

- Proposed by (Hinton et al, 2012)



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

Dropout training

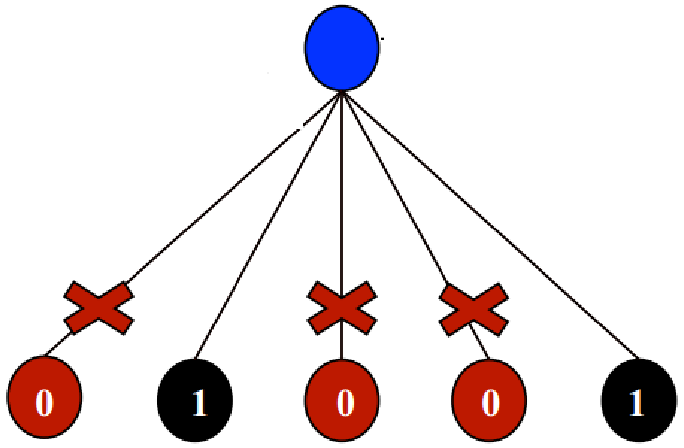


– Dropout of 50% of the hidden units and 20% of the input units (Hinton et al, 2012)

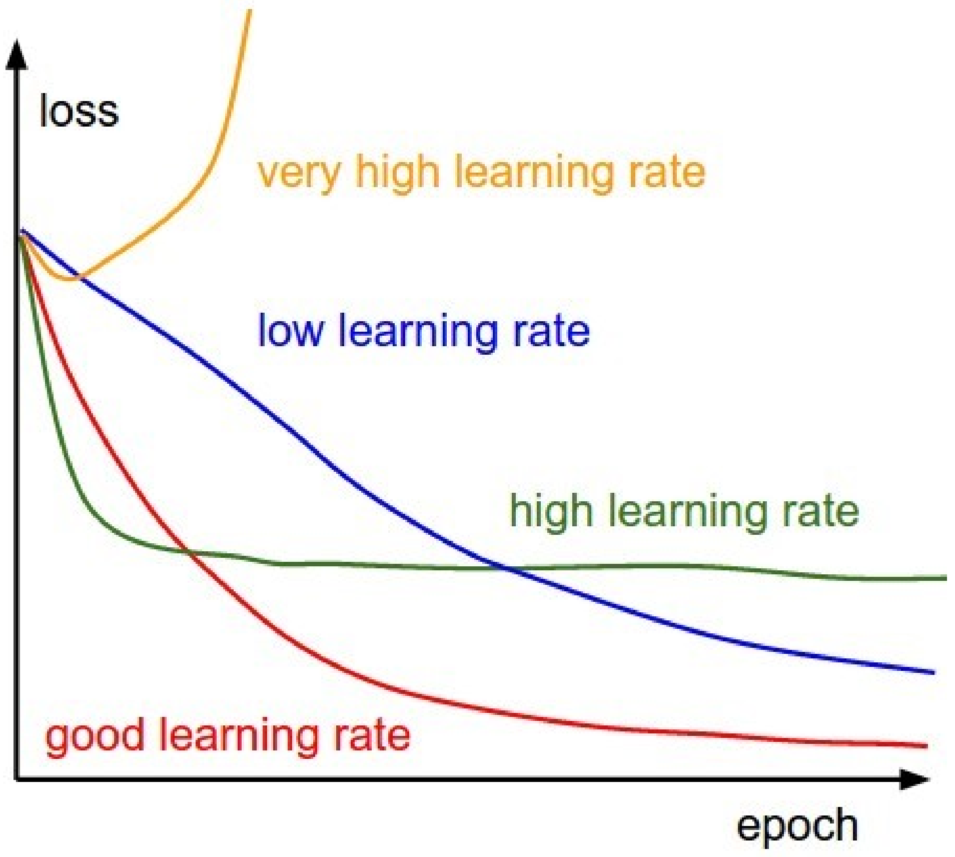
Dropout training

- Model averaging effect
  - Among 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!



Learning Rates



- Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line). This is because there is too much “energy” in the optimization and the parameters are bouncing around chaotically, unable to settle in a nice spot in the optimization landscape.

Some Notes: 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. [Cybenko 1989; Hornik et al. 1989]
  - 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. [Cybenko 1988]

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

Gradient Checks are useful!

- Allow you to know that there are no bugs in your neural network implementation!

  - Implement your gradient
  - Implement a finite difference computation by looping through the parameters of your network, adding and subtracting a small epsilon (∼10^-4) and estimate derivatives

$$f'(\theta) \approx \frac{f(\theta^{+}) - f(\theta^{-})}{2\epsilon} \qquad \theta^{\pm} = \theta \quad \pm \epsilon$$

  - Compare the two and make sure they are almost the same