



Neural Networks and Deep Learning Part I

Dan Roth,

danroth@seas.upenn.edu | <http://www.cis.upenn.edu/~danroth/> | 461C, 3401 Walnut

Slides were created by Dan Roth (for CIS519/419 at Penn or CS446 at UIUC), or by other authors who have made their ML slides available.

Administration (11/16/20)

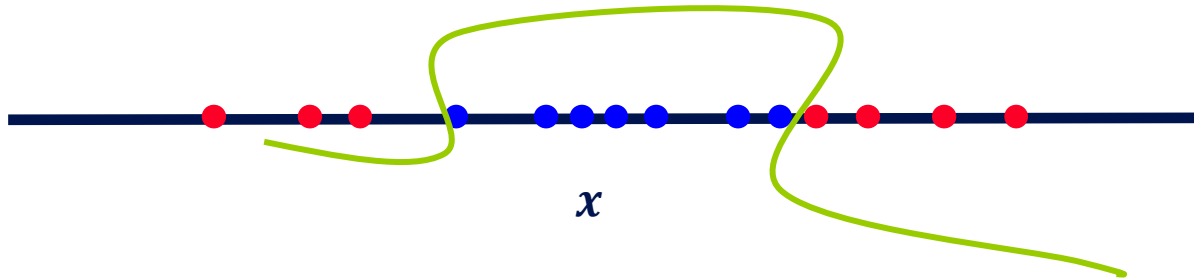
Are we recording? YES!

Available on the web site

- Remember that all the lectures are available on the website **before the class**
 - Go over it and be prepared
 - A new set of written notes will accompany most lectures, with some more details, examples and, (when relevant) some code.
- HW 3: Due Today
- HW4: Out today – NNs, and Bayesian Learning; Due 12/2
 - Recitations will be devoted to introducing you to PyTorch
- Cheating
 - Several problems in HW1 and HW2
- Projects
 - Most of you have chosen a project and a team.

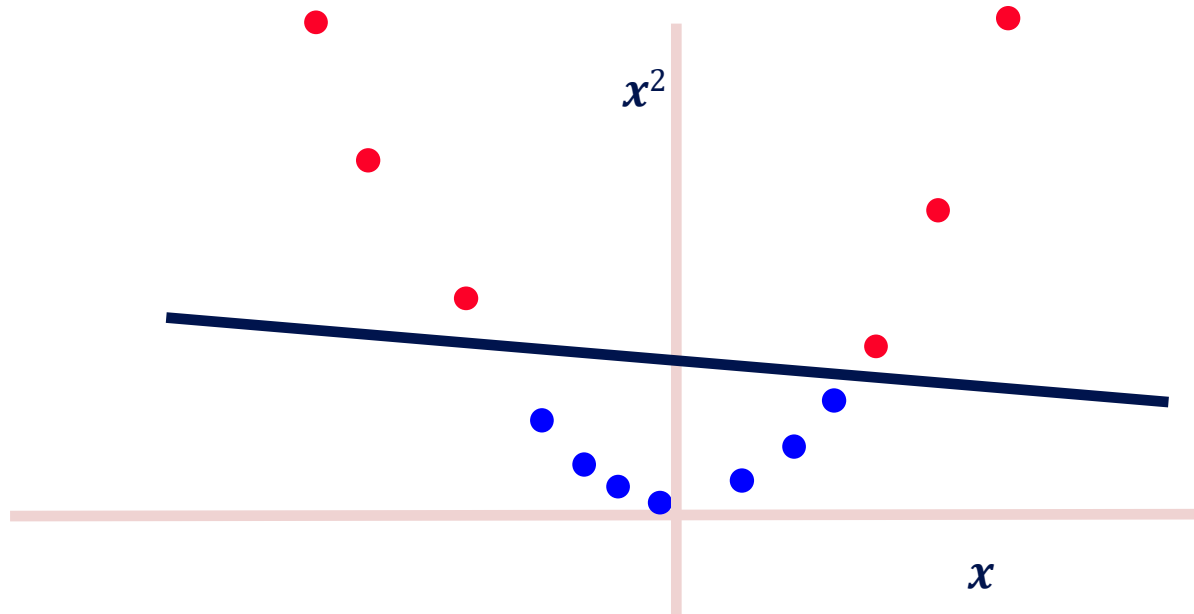
Functions Can be Made Linear

- Data is not linearly separable in one dimension
- Not separable if you insist on using a specific class of functions



Blown Up Feature Space

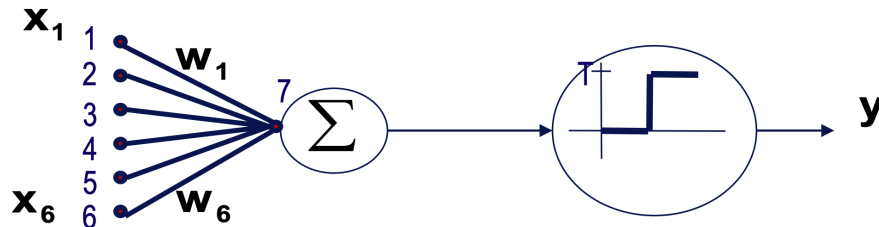
- Data are separable in $\langle x, x^2 \rangle$ space



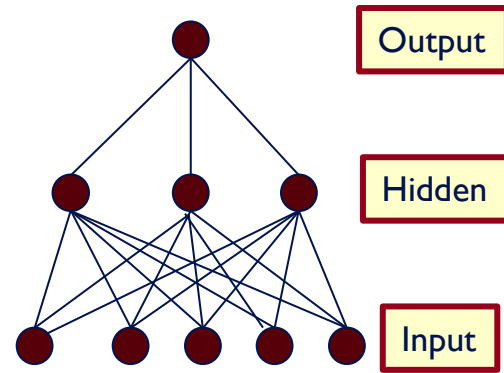
Neural Networks

- Multi-layer networks were designed to overcome the computational (expressivity) limitation of a single threshold element.

Linear Threshold Unit

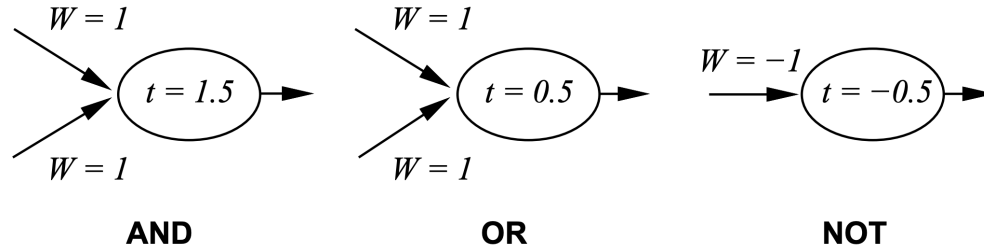


$$y = \text{sign}(\sum w_i x_i - T)$$

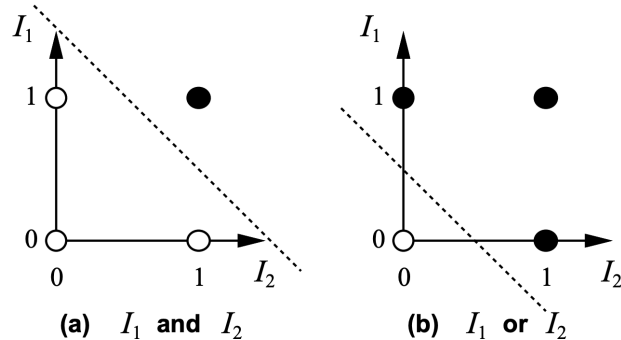


History: Neural Computation

- McCulloch and Pitts (1943) showed how linear threshold units can be used to compute logical functions

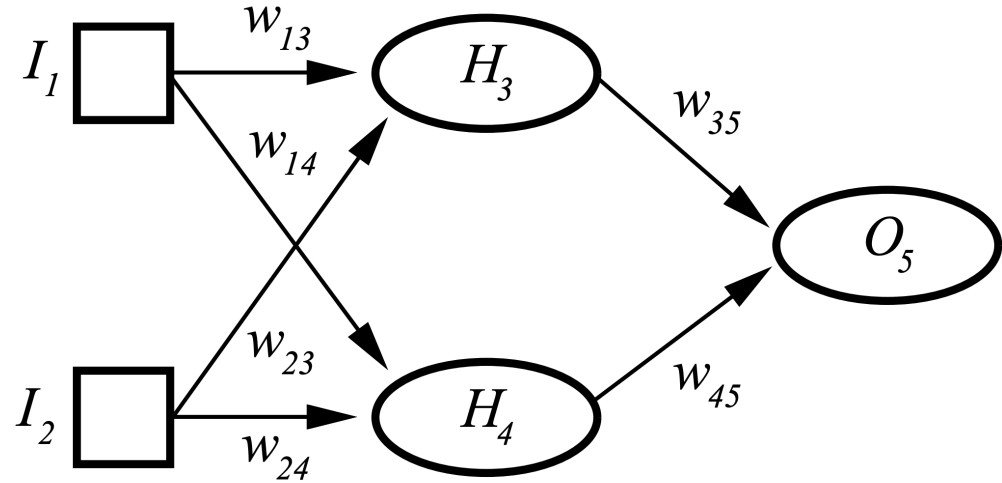
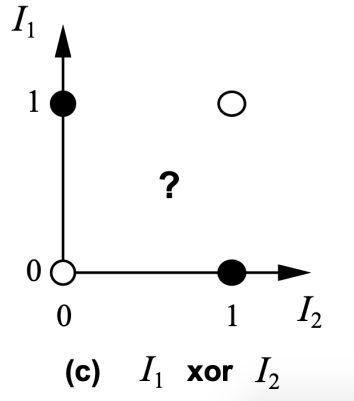


$$y = \text{sign}(\sum w_i I_i - T)$$



History: Neural Computation

- But XOR?

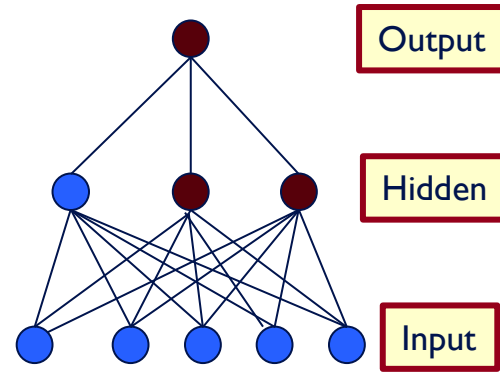
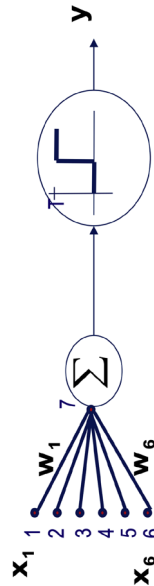


Two Layered Two Unit Network

Neural Networks

- Multi-layer networks were designed to overcome the computational (expressivity) limitation of a single threshold element.

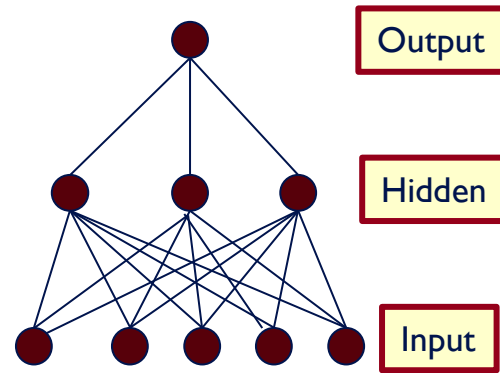
Linear Threshold Unit



Neural Networks

- Multi-layer networks 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.



- Multi-layer networks can represent arbitrary functions, but building effective learning methods for such network was [thought to be] difficult.

Neural Networks

- Neural Networks are functions: $NN: \mathbf{X} \rightarrow Y$
 - where $\mathbf{X} = [0,1]^n$, or \mathbb{R}^n and $Y = [0,1], \{0,1\}$
 - **Robust** approach to approximating **real-valued, discrete-valued and vector valued** target functions.

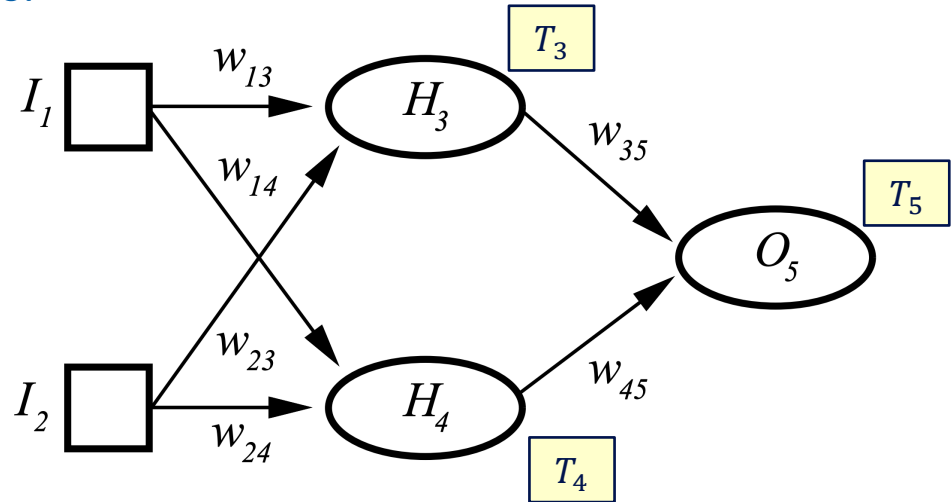
$$H_3 = \text{sign}((w_{13}I_1 + w_{23}I_2) - T_3)$$

$$H_4 = \text{sign}((w_{14}I_1 + w_{24}I_2) - T_4)$$

$$O_5 = \text{sign}((w_{35}H_3 + w_{45}H_4) - T_5)$$

Trainable Parameters:

$$w_{13}, w_{14}, w_{23}, w_{24}, w_{35}, w_{45}, T_3, T_4, T_5$$



Neural Networks

- Neural Networks are functions: $NN: \mathbf{X} \rightarrow Y$
 - where $\mathbf{X} = [0,1]^n$, or \mathbb{R}^n and $Y = [0,1], \{0,1\}$
 - 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.
- Learning:
 - The Backpropagation algorithm for neural networks has been shown successful in many practical problems

Motivation for Neural Networks

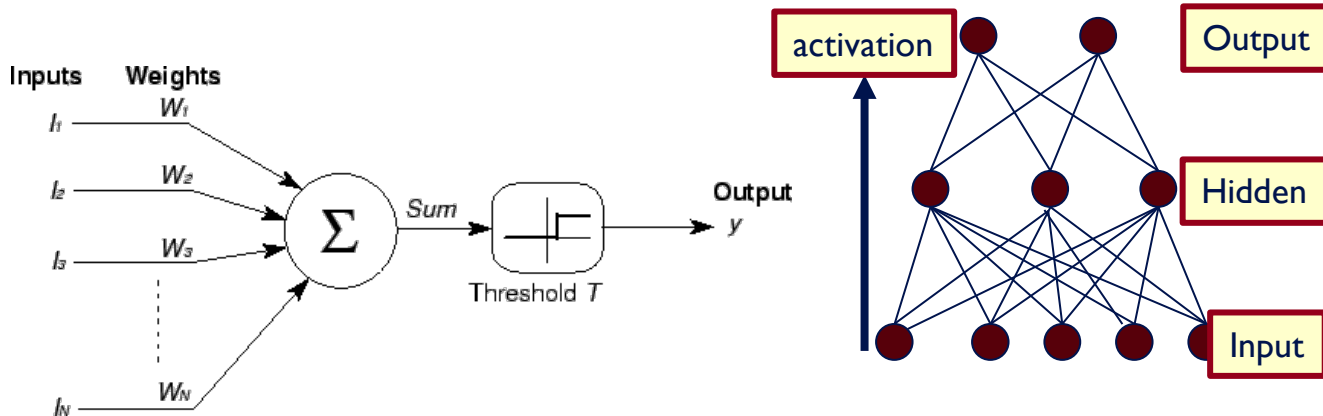
- Inspired by biological neural network systems
 - But are not identical to them
- We are currently on rising part of a wave of interest in NN architectures, after a long downtime from the mid-90-ies.
 - Better computer architecture (parallelism on GPUs & TPUs)
 - A lot more data than before; in many domains, supervision is available.

Motivation for Neural Networks

- One potentially interesting perspective:
 - We used to think about NNs only as function approximators.
 - Geoffrey Hinton introduced “Restricted Boltzman Machines” (RBMs) in the mid 2000s – method to learn high-level representations of input
 - Many other ideas focusing in the [Intermediate Representations](#) of NNs
 - Ideas are being developed on the value of these intermediate representations for transfer learning, for the meaning they represent etc.
- We will present in the next two lectures a few of the basic architectures and learning algorithms, and provide some examples for applications

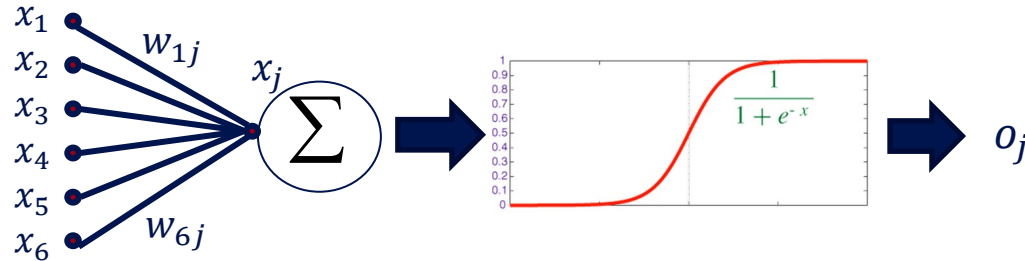
Basic Unit in Multi-Layer Neural Network

- Threshold units: $o_j = \text{sgn}(\mathbf{w} \cdot \mathbf{x} - T)$ introduce non-linearity
 - But not differentiable,
 - Hence unsuitable for learning via Gradient Descent



Logistic Neuron / Sigmoid Activation

- 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: $net_j = \sum w_{ij} \cdot x_i$

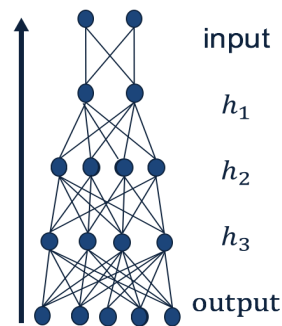
- Output of a unit is defined as: $o_j = \sigma(net_j) = \frac{1}{1 + \exp(-(net_j - T_j))}$

Representational Power

- 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 functions** from which arbitrary function can be composed.
- **Any function** can be approximated to arbitrary accuracy by a **three layer** network.

Quiz Time!

- Given a neural network, how can we make predictions?
 - Given input, calculate the output of each layer (starting from the first layer), until you get to the output.
- What is required to fully specify a neural network?
 - The weights.
- Why NN predictions can be quick?
 - Because many of the computations could be parallelized.
- What makes a neural networks expressive?
 - The non-linear units.



Training a Neural Net

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>

Two layer Two Unit Neural Network

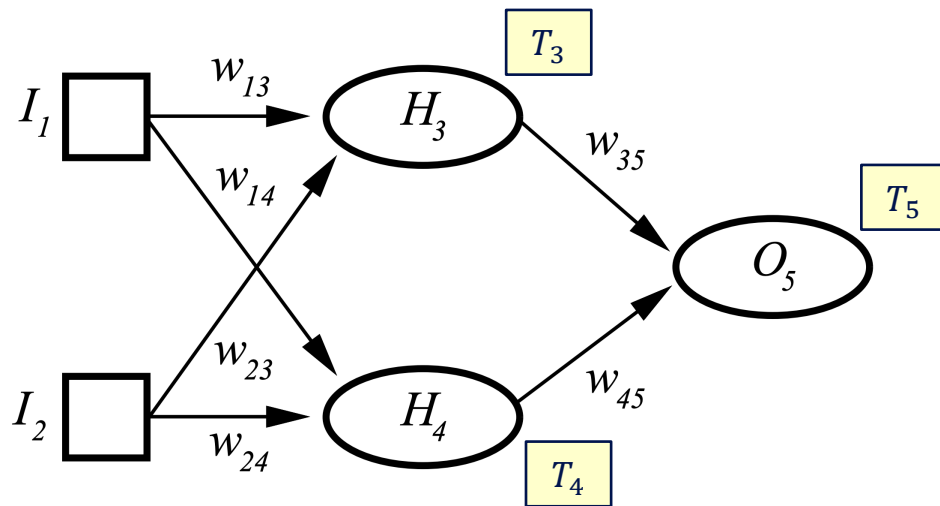
$$H_3 = \sigma((w_{13}I_1 + w_{23}I_2) - T_3)$$

$$H_4 = \sigma((w_{14}I_1 + w_{24}I_2) - T_4)$$

$$O_5 = \sigma((w_{35}H_3 + w_{45}H_4) - T_5)$$

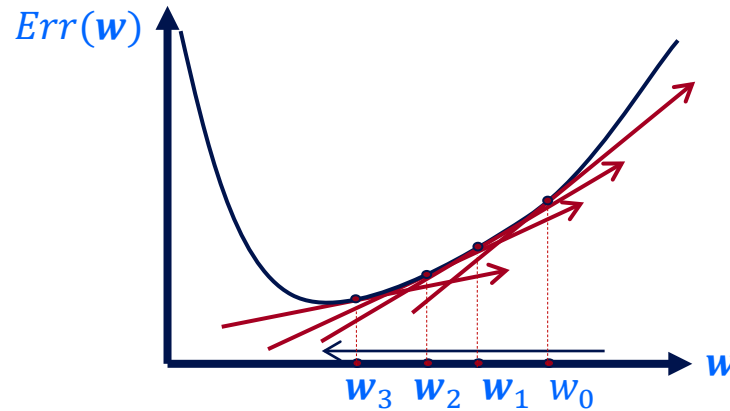
Trainable Parameters:

$$w_{13}, w_{14}, w_{23}, w_{24}, w_{35}, w_{45}, T_3, T_4, T_5$$



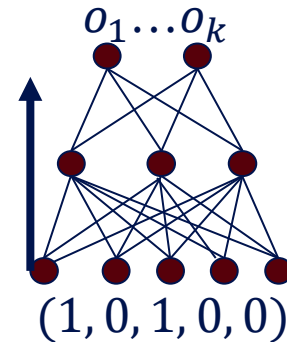
Gradient Descent

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



Backpropagation Learning Rule

- Since there could be multiple output units, we define the error as the sum over all the network output units.
- $Err(\mathbf{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in K} (t_{kd} - o_{kd})^2$
 - where D is the set of training examples,
 - K is the set of output units

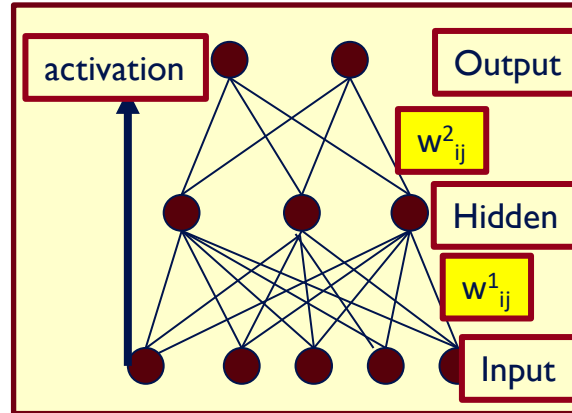


- This is used to derive the (global) learning rule which performs gradient descent in the weight space in an attempt to minimize the error function.

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

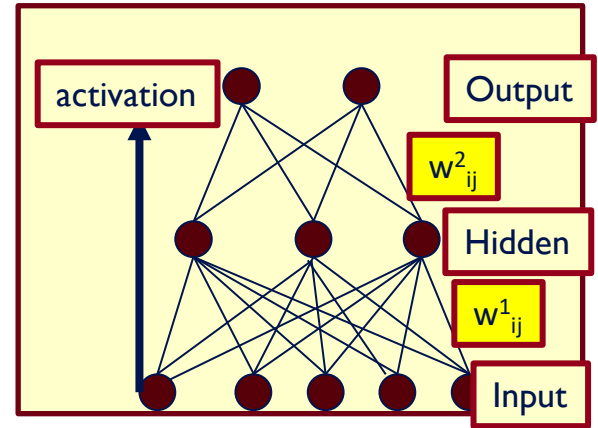
Learning with a Multi-Layer Perceptron



- 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**.





What is the chain rule for differentiating a composite function? $d/dx(f(g(x))) =$

Chain Rule

$$\frac{d}{dx} [f(g(x))] = f'(g(x))g'(x)$$

$$\frac{df}{dx} = \frac{df}{dg} \frac{dg}{dx}$$

Some facts from real analysis

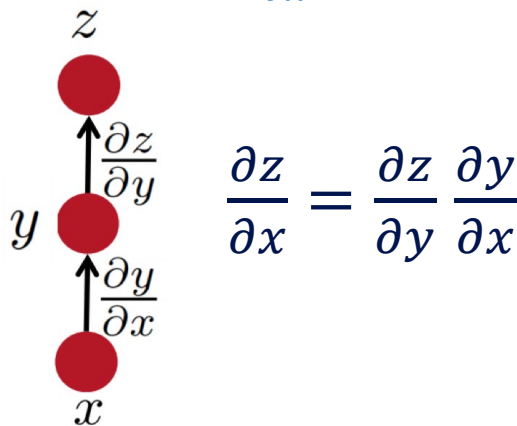
- First let's get the notation right:
- The arrow shows functional dependence of z on y
 - i.e. given y , we can calculate z .
 - e.g., for example: $z(y) = 2y^2$

The derivative of z , with respect to y .



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



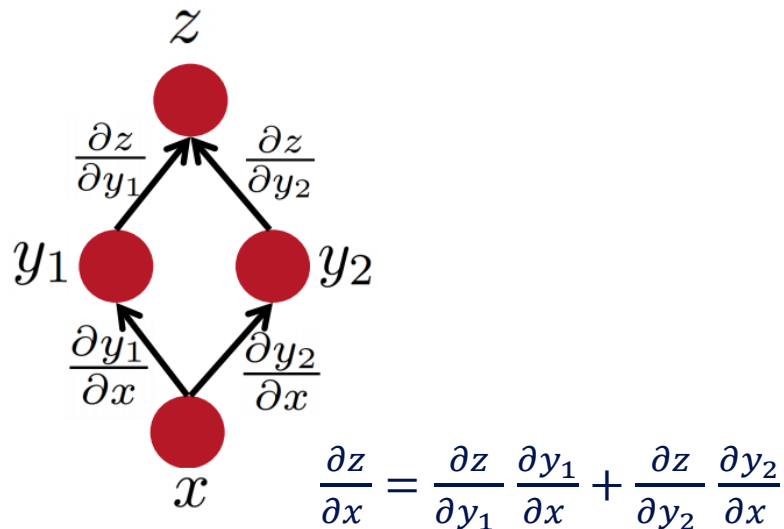
We will use these facts to derive the details of the Backpropagation algorithm.

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.

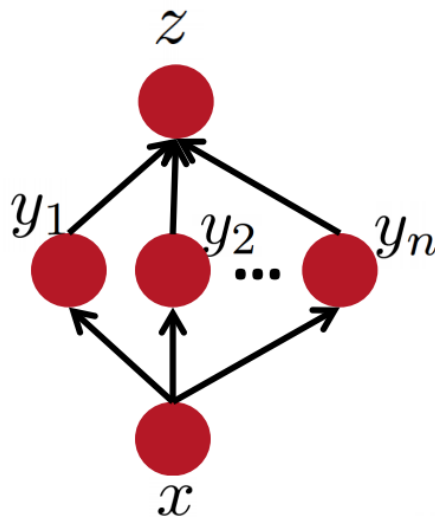
Some facts from real analysis

- Multiple path chain rule



Some facts from real analysis

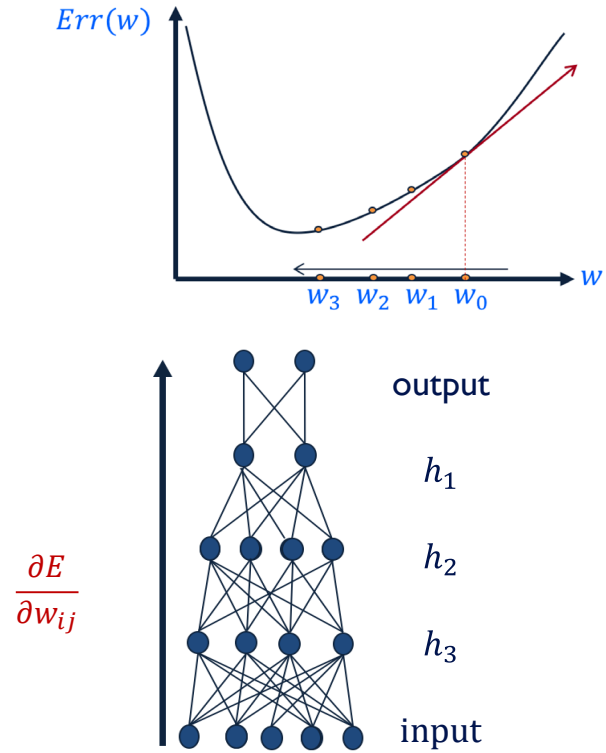
- Multiple path chain rule: general



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

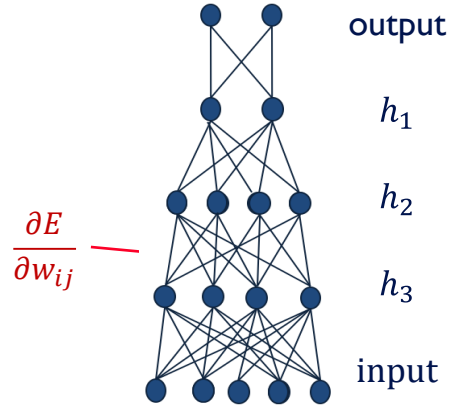
Key Intuitions Required for BP

- Gradient Descent
 - Change the weights in the direction of gradient to minimize the error function.
- Chain Rule
 - Use the chain rule to calculate the weights of the intermediate weights
- Dynamic Programming (Memoization)
 - Memoize the weight updates to make the updates faster.



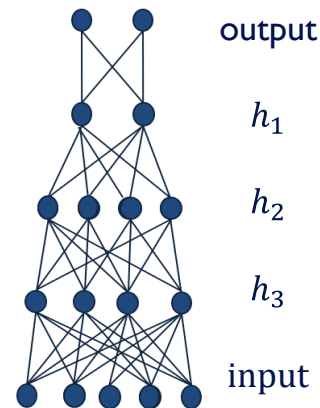
Backpropagation: the big picture

- Loop over instances:
 1. The forward step
 - Given the input, make predictions layer-by-layer, starting from the first layer)
 2. The backward step
 - Calculate the error in the output
 - Update the weights layer-by-layer, starting from the final layer



Quiz time!

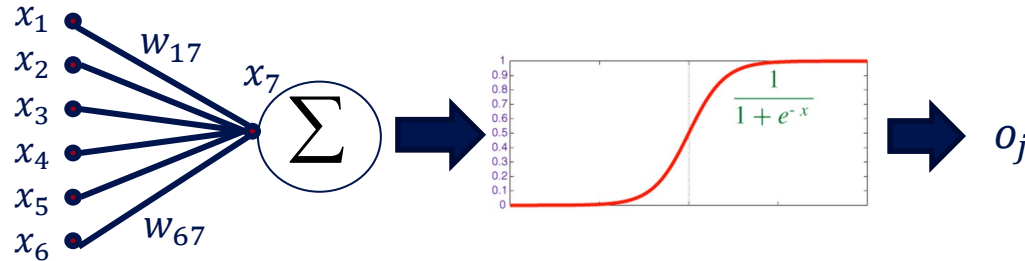
- What is the purpose of forward step?
 - To make predictions, given an input.
- What is the purpose of backward step?
 - To update the weights, given an output error.
- Why do we use the chain rule?
 - To calculate gradient in the intermediate layers.
- Why backpropagation could be efficient?
 - Because it can be parallelized.



Deriving the update rules

Reminder: Model Neuron (Logistic)

- 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:
- Output of a unit is defined as:

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

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

Note:

Other gates, beyond Sigmoid, can be used (TanH, ReLu)
Other Loss functions, beyond LMS, can be used.

Derivation of Learning Rule

- The weights are updated incrementally; the error is computed for each example and the weight update is then derived.

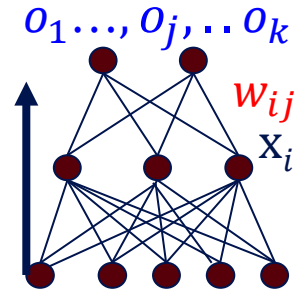
$$E_d(\mathbf{w}) = \frac{1}{2} \sum_{k \in K} (t_k - o_k)^2$$

- w_{ij} influences the output o_j only through net_j

$$o_j = \frac{1}{1 + \exp\{-(\text{net}_j - T)\}} \quad \text{and} \quad \text{net}_j = \sum w_{ij} \cdot x_i$$

- Therefore:

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



Derivatives

- Function 1 (error):

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

- $\frac{\partial E}{\partial o_i} = -(t_i - o_i)$

- Function 2 (linear gate):

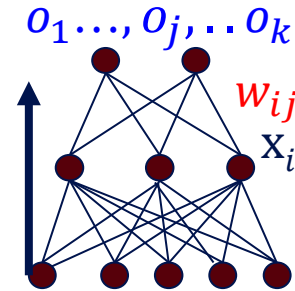
- $\text{net}_j = \sum w_{ij} \cdot x_i$

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

- Function 3 (differentiable activation function):

- $o_i = \frac{1}{1 + \exp\{-(\text{net}_j - T)\}}$

- $\frac{\partial o_i}{\partial \text{net}_j} = \frac{\exp\{-(\text{net}_j - T)\}}{(1 + \exp\{-(\text{net}_j - T)\})^2} = o_i(1 - o_i)$



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

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

$$\frac{d\sigma(x)}{d(x)} = \sigma(x) \cdot (1 - \sigma(x))$$

Derivation of Learning Rule (2)

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

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

$$\frac{\partial o_j}{\partial net_j} = o_j(1 - o_j)$$

$$= \frac{1}{1 + \exp\{-(net_j - T_j)\}}$$

$$\sum w_{ij} \cdot x_i$$

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

$$= -(t_j - o_j) o_j(1 - o_j) x_i$$

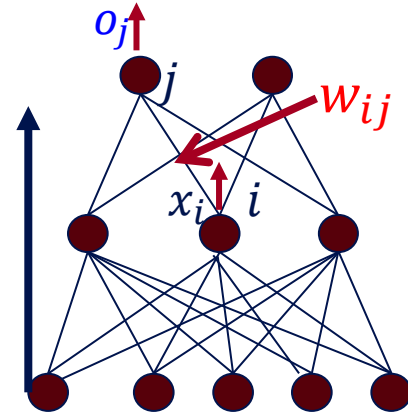
Derivation of Learning Rule (3)

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

$$\begin{aligned}\Delta w_{ij} &= R(t_j - o_j)o_j(1 - o_j)x_i \\ &= R\delta_j x_i\end{aligned}$$

Where we defined:

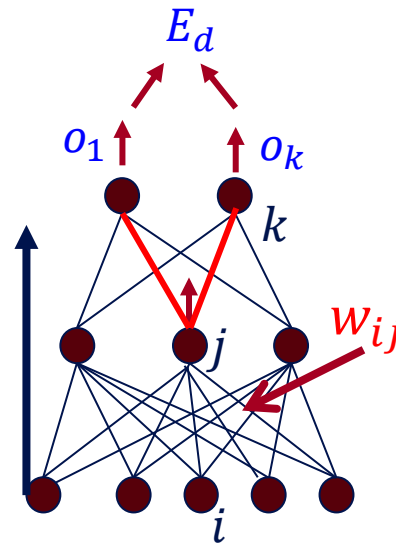
$$\delta_j = \frac{\partial E_d}{\partial \text{net}_j} = (t_j - o_j)o_j(1 - o_j)$$



Derivation of Learning Rule (4)

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

$$\frac{\partial E_d}{\partial w_{ij}}$$

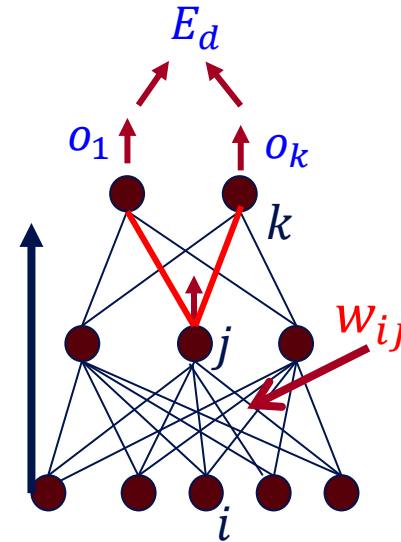


Derivation of Learning Rule (4)

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

$$\begin{aligned}
 \frac{\partial E_d}{\partial w_{ij}} &= \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} \\
 &= \frac{\partial E_d}{\partial \text{net}_j} x_i = \sum_{k \in \text{parent}(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i \\
 &= \sum_{k \in \text{parent}(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i
 \end{aligned}$$

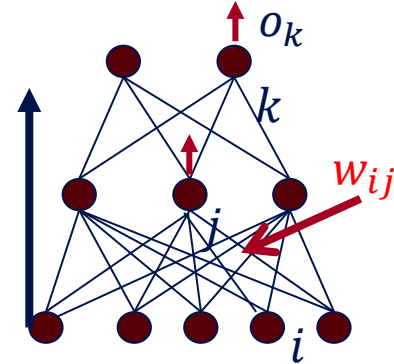
$\text{net}_j = \sum w_{ij} \cdot x_i$



Derivation of Learning Rule (5)

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

$$\begin{aligned}\frac{\partial E_d}{\partial w_{ij}} &= \sum_{k \in \text{parent}(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i = \\ &= \sum_{k \in \text{parent}(j)} -\delta_k \frac{\partial \text{net}_k}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} x_i \\ &= \sum_{k \in \text{parent}(j)} -\delta_k w_{jk} o_j (1 - o_j) x_i\end{aligned}$$



Derivation of Learning Rule (6)

- Weights of hidden units:

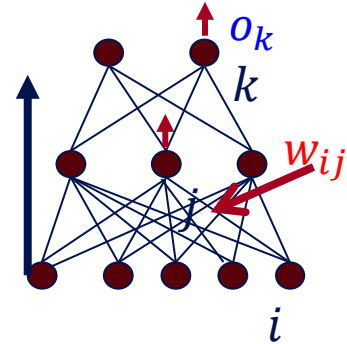
– w_{ij} is changed by:

$$\begin{aligned}\Delta w_{ij} &= R o_j (1 - o_j) \cdot \left(\sum_{k \in \text{parent}(j)} -\delta_k w_{jk} \right) x_i \\ &= R \delta_j x_i\end{aligned}$$

- Where

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

- 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:

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

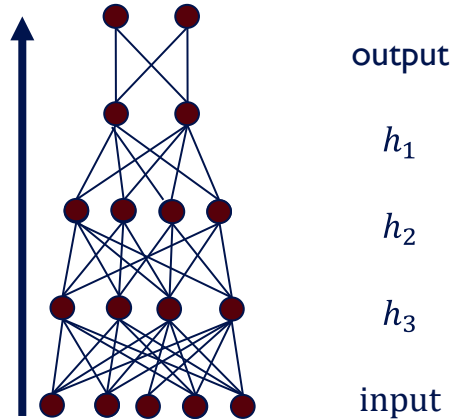
$$\delta_j = o_j(1 - o_j) \cdot \sum_{k \in \text{downstream}(j)} -\delta_k w_{jk}$$

1. For each hidden unit, compute error term: $\Delta w_{ij} = R \delta_j x_i$
2. Update network weights with Δw_{ij}

End epoch

More Hidden Layers

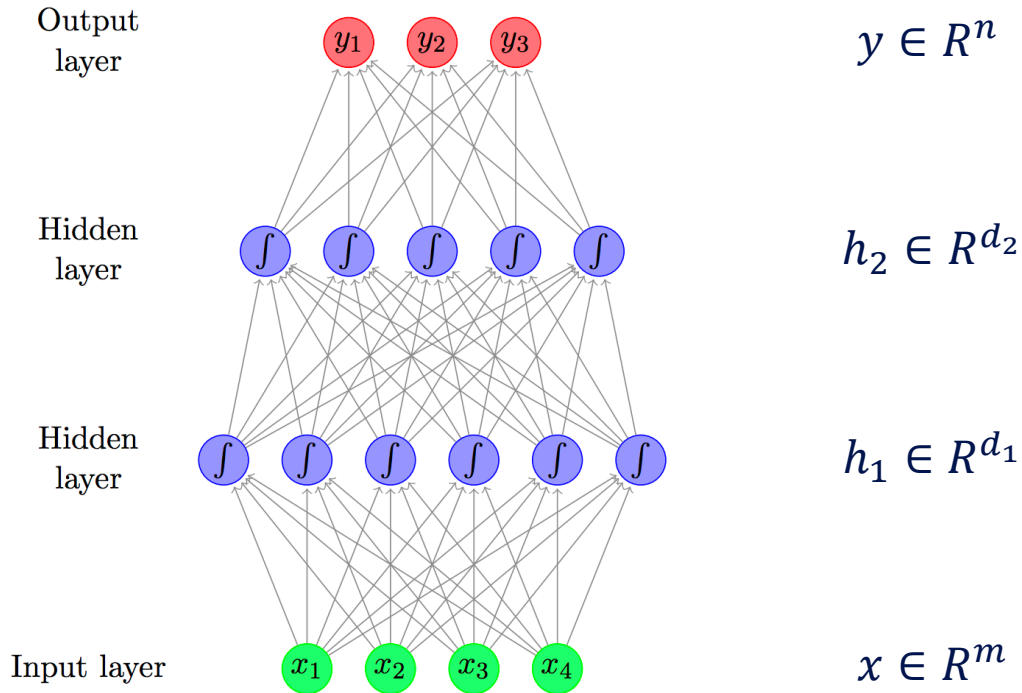
- The same algorithm holds for more hidden layers.



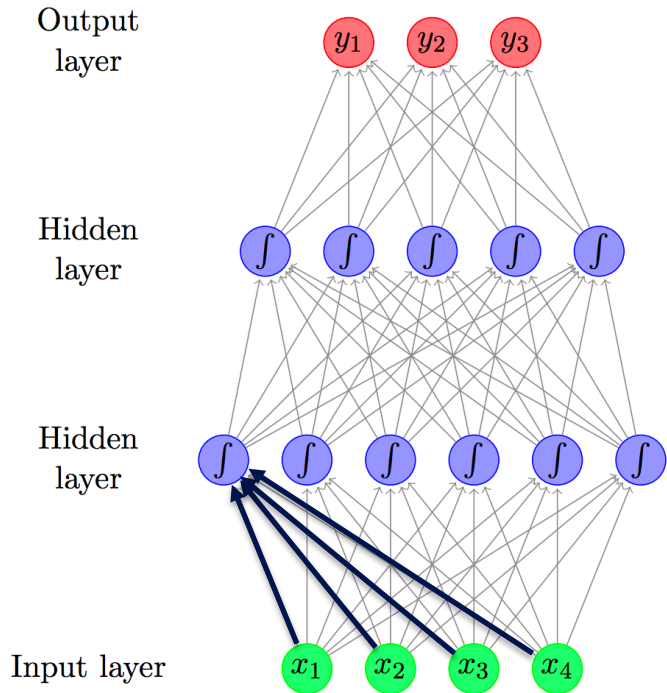
Demo time!

- Link: <https://playground.tensorflow.org/>

Feed-forward (FF) Network / Multi-layer Perceptron (MLP)



Feed-forward (FF) Network / Multi-layer Perceptron (MLP)



$$y \in R^n$$

$$h_2 \in R^{d_2}$$

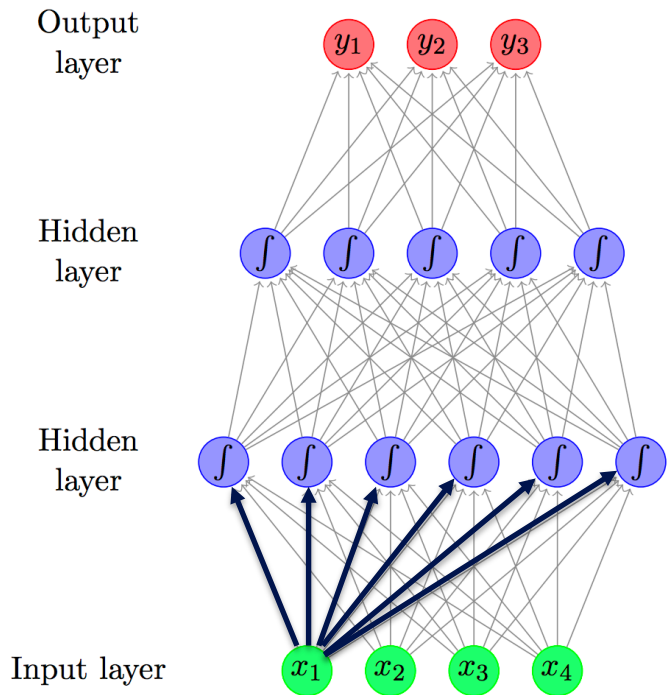
$$h_1 \in R^{d_1}$$

$$h_1 = \sigma(W_1 x) ; W_1 \in R^{d_1 \times m}$$

$$x \in R^m$$

w_{11}	w_{21}	w_{31}	w_{41}

Feed-forward (FF) Network / Multi-layer Perceptron (MLP)



$$y \in R^n$$

$$h_2 \in R^{d_2}$$

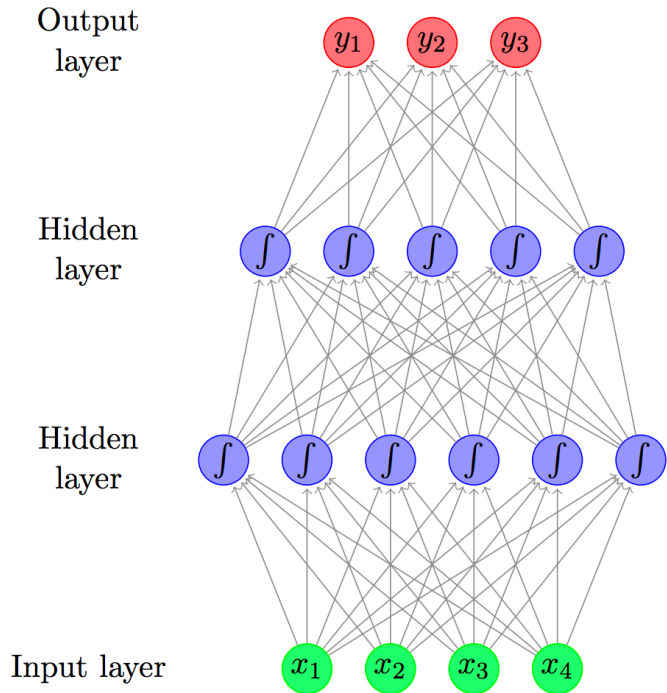
$$h_1 \in R^{d_1}$$

$$h_1 = \sigma(W_1 x) ; W_1 \in R^{d_1 \times m}$$

$$x \in R^m$$

w_{11}			
w_{12}			
w_{13}			
w_{14}			
w_{15}			
w_{16}			

Feed-forward (FF) Network / Multi-layer Perceptron (MLP)



$$y \in R^n \quad y = \sigma(W_3 h_2) \quad ; \quad W_3 \in R^{n \times d_2}$$

$$h_2 \in R^{d_2} \quad h_2 = \sigma(W_2 h_1) \quad ; \quad W_2 \in R^{d_2 \times d_1}$$

$$h_1 \in R^{d_1} \quad h_1 = \sigma(W_1 x) \quad ; \quad W_1 \in R^{d_1 \times m}$$

$$x \in R^m$$

The Backpropagation Algorithm

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

For each example (x_i, t_i) in the training set do:

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

$$E = \sum (t_i^k - o_i^k)^2$$

3. Compute the gradient for all weight values, Δw_{ij}
4. Update network weights with $w_{ij} = w_{ij} - R * \Delta w_{ij}$

End epoch

Auto-differentiation packages such as Tensorflow, Torch, etc. help!

[Quick example in code](#)

Comments on Training

- **No guarantee of convergence**; neural networks form non-convex functions with multiple 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

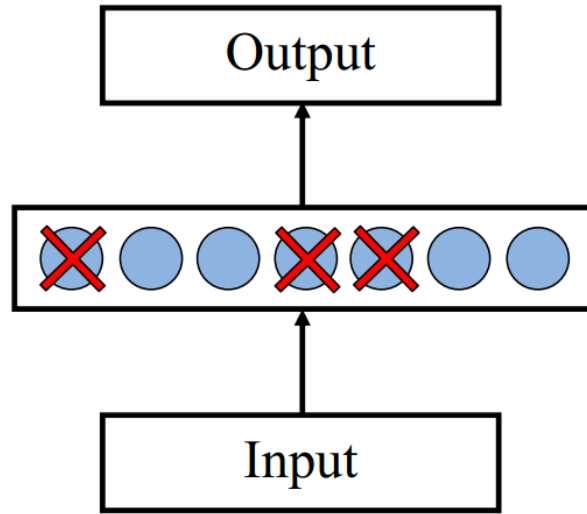
- Running too many epochs and/or a NN with many hidden layers may lead to an **overfit** network
- Keep an **held-out validation** set and test accuracy after every epoch
- Early stopping: maintain weights for best performing network on the validation set and return it when performance decreases significantly 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)

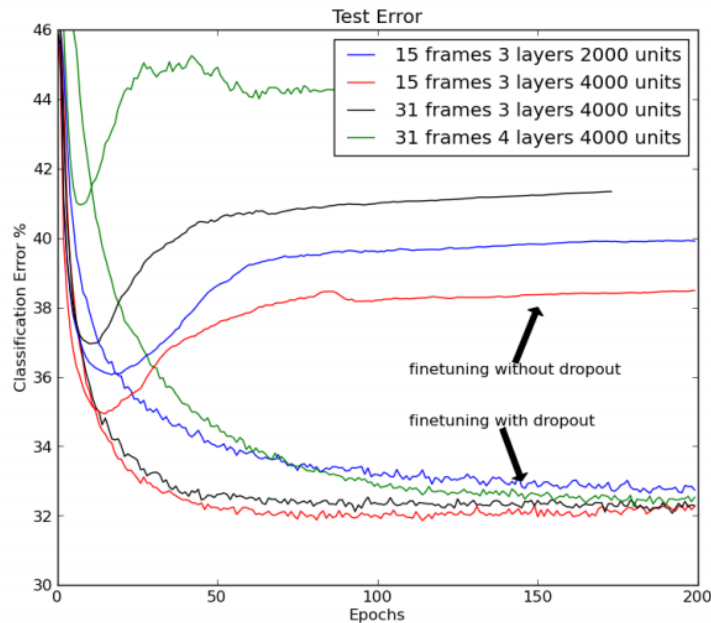
Dropout training

- Proposed by (Hinton et al, 2012)



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

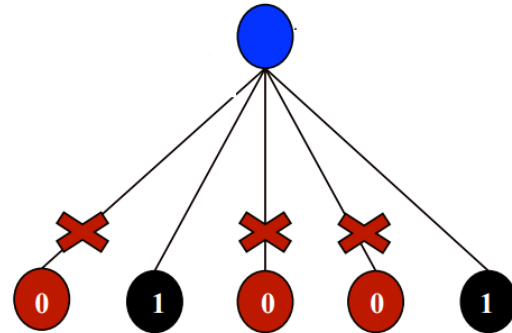
Dropout training



- 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!



Recap: Multi-Layer Perceptrons

- Multi-layer network
 - A global approximator
 - Different rules for training it
- The Back-propagation
 - Forward step
 - Back propagation of errors

- Congrats! Now you know the most important algorithm in neural networks!

- Next Time:
 - Convolutional Neural Networks
 - Recurrent Neural Networks

