CIS 519/419
Applied Machine Learning

www.seas.upenn.edu/~cis519

Dan Roth

danroth@seas.upenn.edu

http://www.cis.upenn.edu/~danroth/

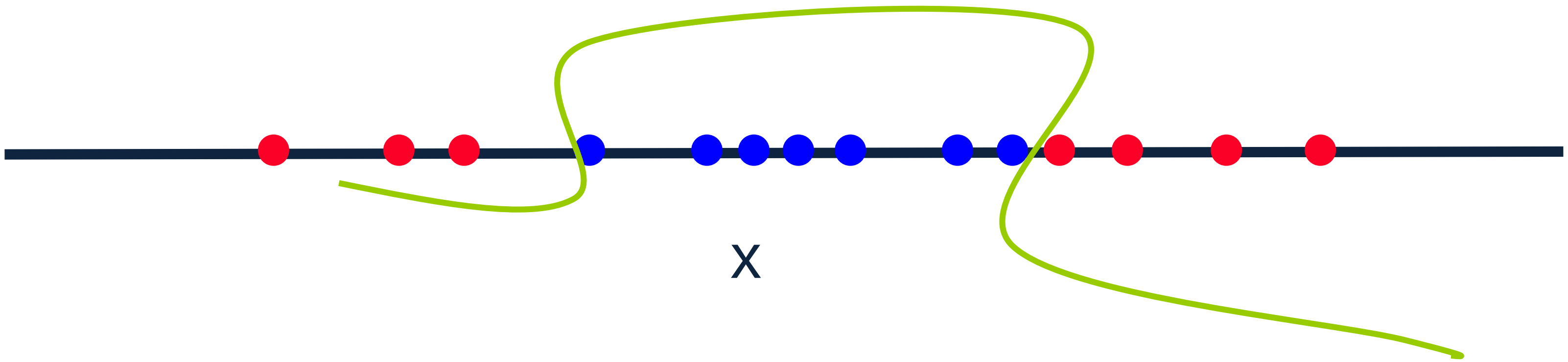
461C, 3401 Walnut

Lecture given by Daniel Khashabi

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

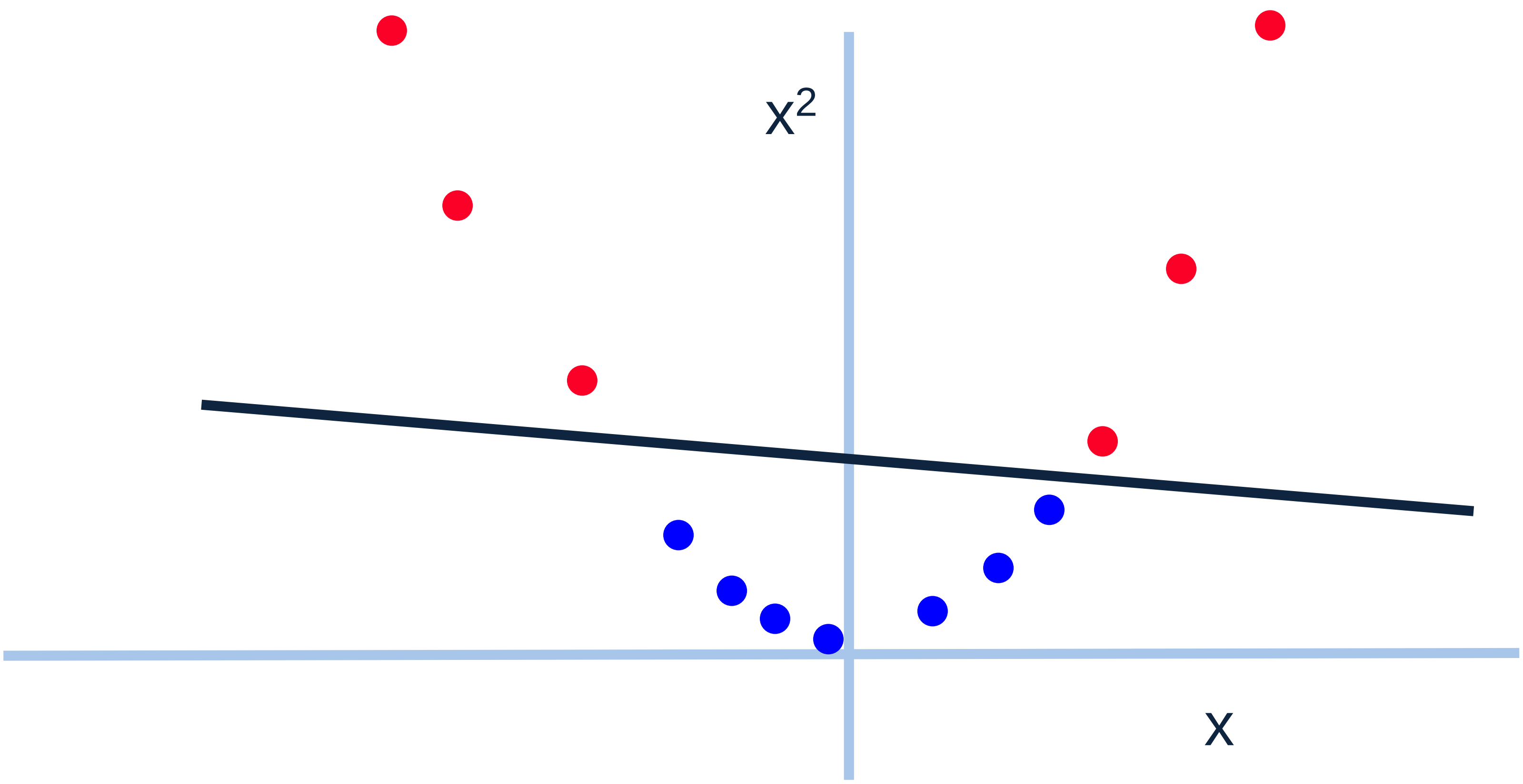
Functions Can be Made Linear

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



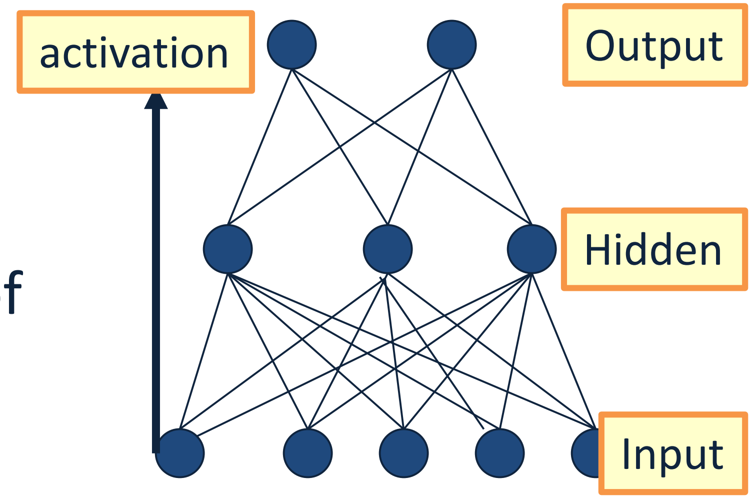
Blown Up Feature Space

- Data are separable in <x, x^2> space



Multi-Layer Neural Network

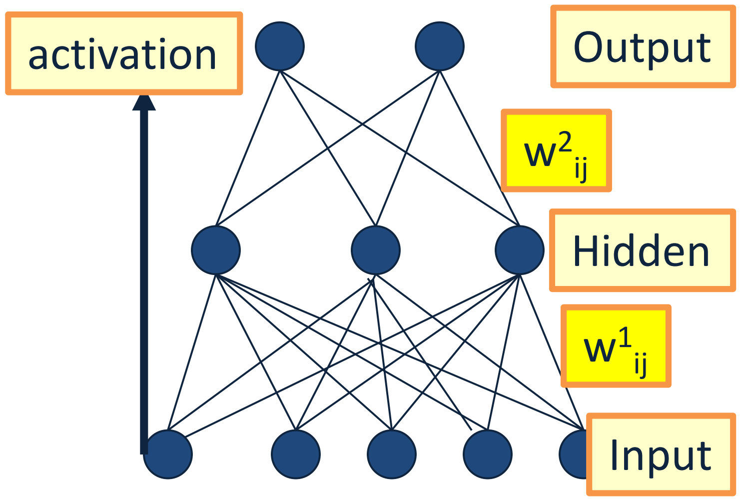
- Multi-layer network were designed to overcome the computational (expressivity) limitation of a single threshold element.
- The idea is to stack several layers of threshold elements, each layer using the output of the previous layer as input.



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

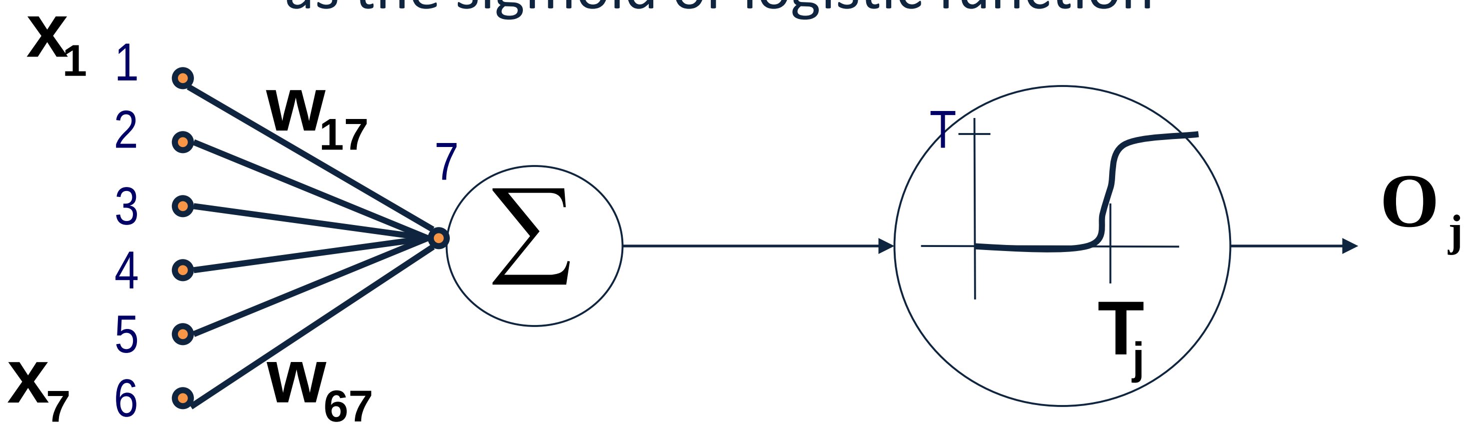
Basic Units

- Linear Unit: Multiple layers of linear functions o_j = w ¢x produce linear functions. We want to represent nonlinear functions.
- Need to do it in a way that facilitates learning
- Threshold units: o_j = sgn(w ¢x) are not differentiable, hence unsuitable for gradient descent.
- The key idea was to notice that the discontinuity of the threshold element can be represents by a smooth non-linear approximation: o_j = [1+ exp{-w ¢x}]$^{-1}$
- (Rumelhart, Hinton, Williiam, 1986), (Linnainmaa, 1970) , see: http://people.idsia.ch/~juergen/who-invented-backpropagation.html)



Model Neuron (Logistic)

- Us a non-linear, differentiable output function such as the sigmoid or logistic function

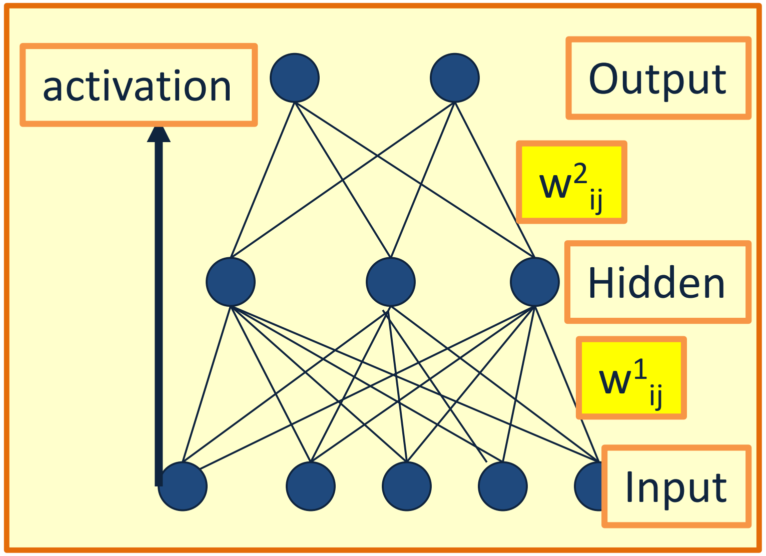


- Net input to a unit is defined as: $\mathbf{net_j = \sum w_{ij} \bullet x_i}$
- Output of a unit is defined as:

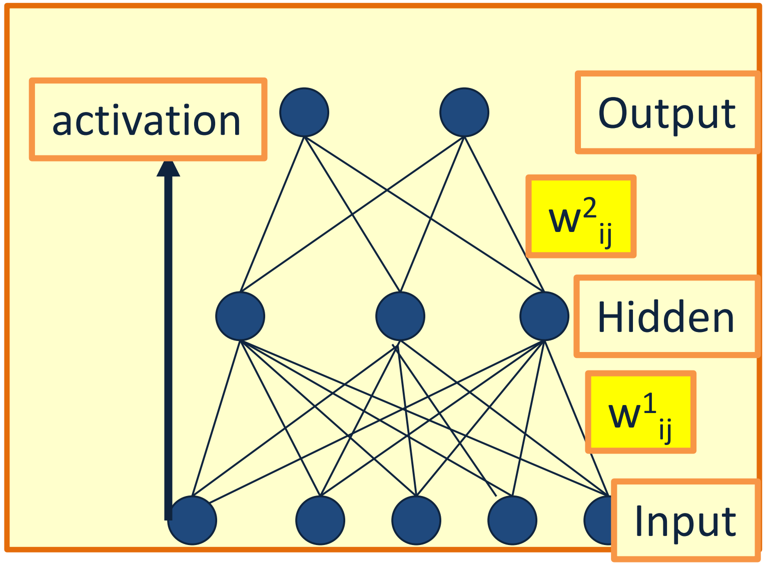
$$\mathbf{O_j = \frac{1}{1 + e^{-(net_j - T_j)}}}$$

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 (multiple options) that is a differentiable function of the output, that 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. This can be done, for example, using gradient descent .
- This results in an algorithm called back-propagation.

Neural Networks

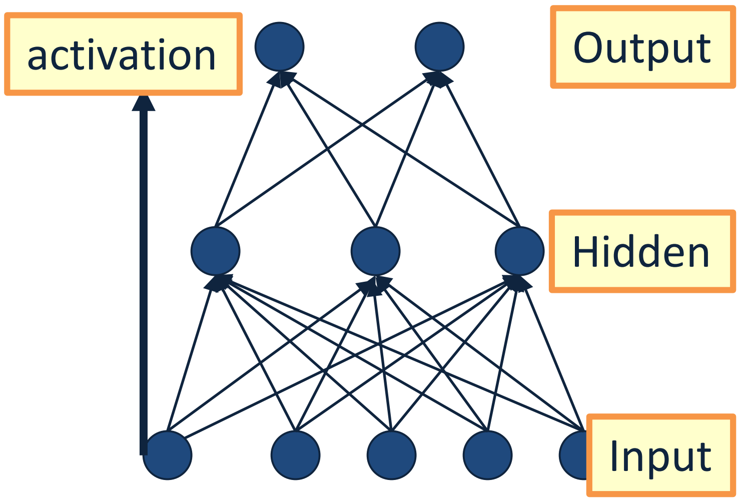
- **Robust** approach to approximating **real-valued**, **discrete-valued** and **vector valued** target functions.
- Among the most effective **general purpose** supervised learning method currently known.
- Effective especially for **complex and hard to interpret input data** such as real-world sensory 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

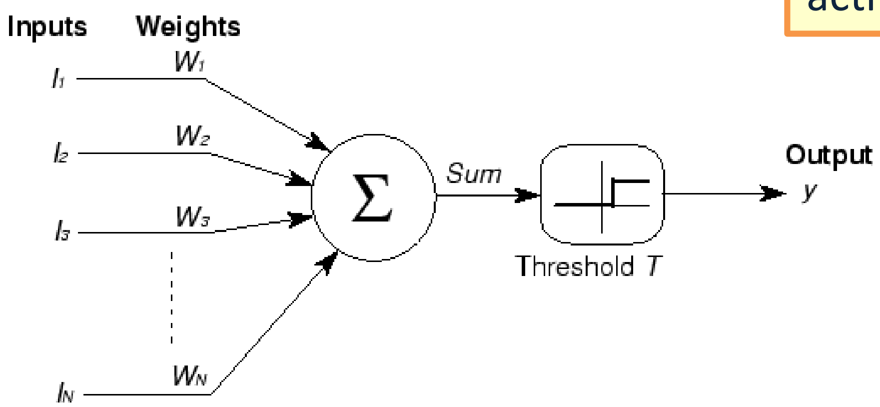
Motivation for Neural Networks

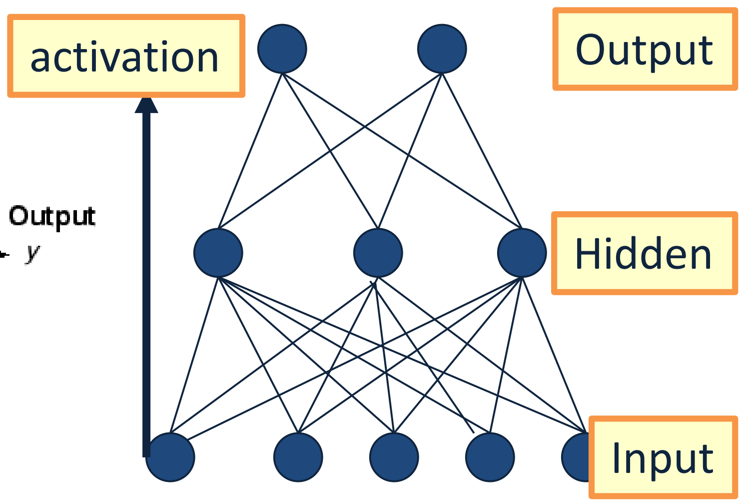
- Minimal to no algorithmic changes
- One potentially interesting perspective:
  - Before we looked at NN only as function approximators.
  - Now, we look at the intermediate representations generated while learning as meaningful
  - Ideas are being developed on the value of these intermediate representations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

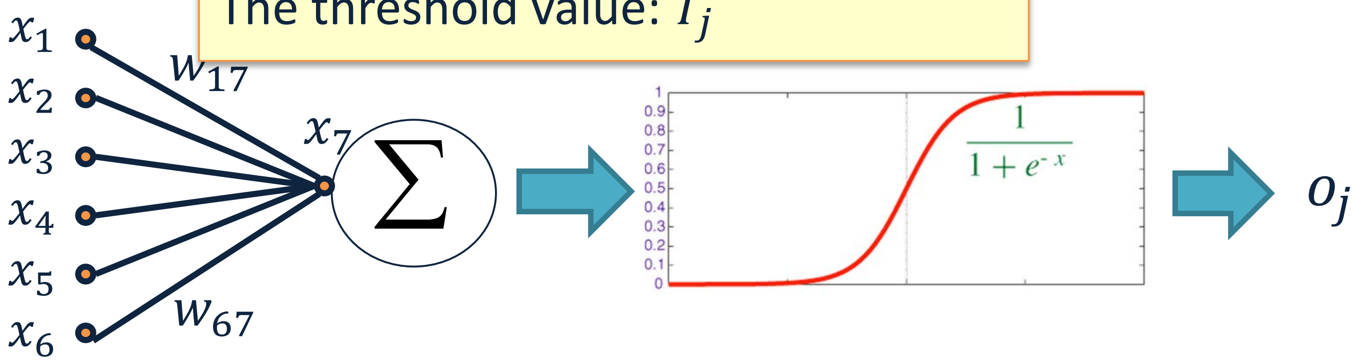




Model Neuron (Logistic)

- Neuron is mode eighted
  links w_{ij} to oth

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



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

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
- 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

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

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

History: Learning Rules

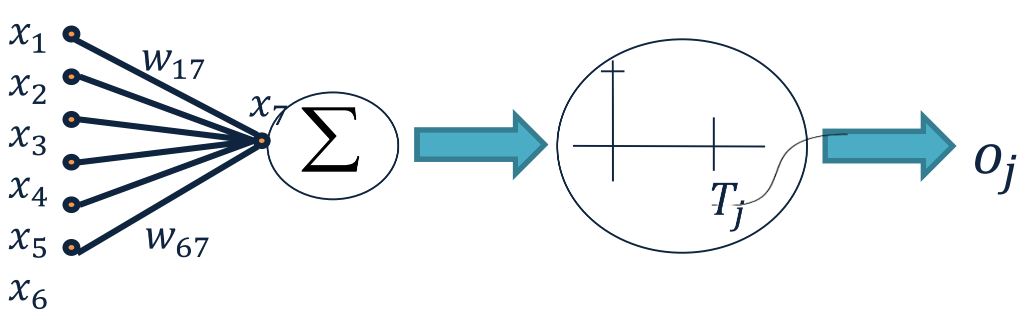
- Hebb (1949) suggested that if two units are both active (firing) then the weights between them should increase:

$$w_{ij} = w_{ij} + R o_i o_j$$

  - 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

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 $w_{ij} \leftarrow w_{ij} + R(t_j - o_j)x_i$
  - 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

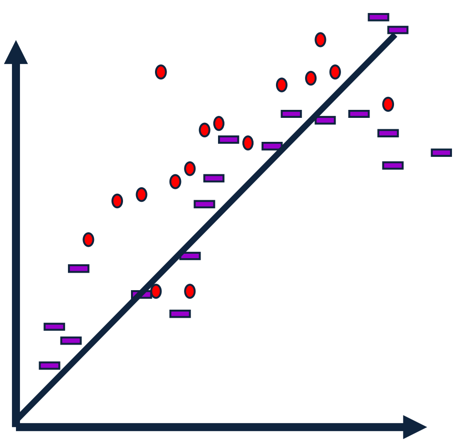
- This incremental update rule provides an approximation to the goal:
  - Find the best linear approximation of the data

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

  - where:

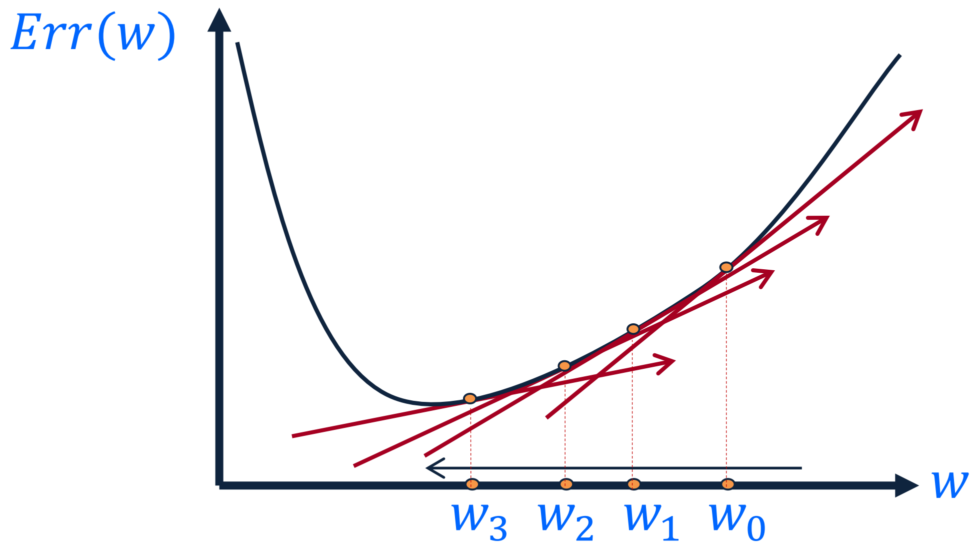
$$o_d = \sum_i w_{ij} . x_i = \vec{w}^{(j)} . \vec{x}$$

    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

General Stochastic Gradient Algorithms

Learning rate

gradient

The loss Q: a function of x, w and y

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

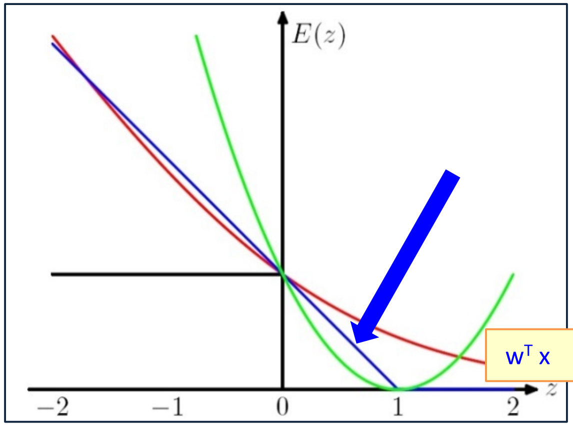
- LMS: $Q((x, y), w) = 1/2\,(y - w^T x)^2$
- leads to the update rule (Also called Widrow's Adaline):

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

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

- Another common loss function is:
- Hinge loss:

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



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

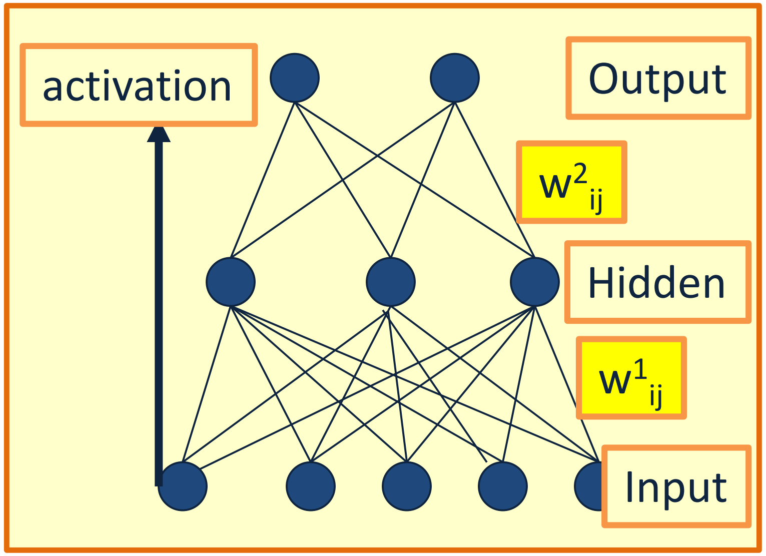
Here g = -yx
Good to think about the case of Boolean examples

Summary: Single Layer Network

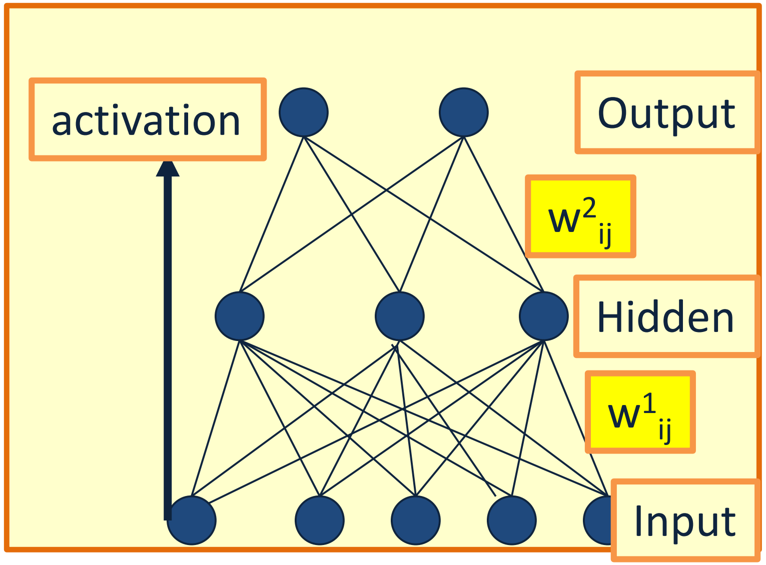
- Variety of update rules
  - Multiplicative
  - Additive
- **Batch** and **incremental** algorithms
- Various convergence and efficiency conditions
- There are other ways to learn linear functions
  - Linear Programming (general purpose)
  - Probabilistic Classifiers (some assumption)
- Key algorithms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



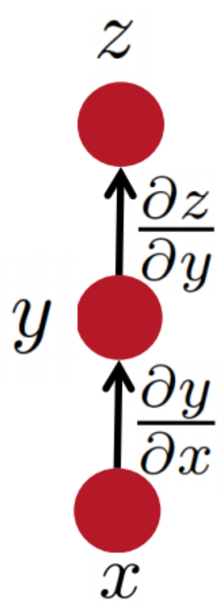
Learning with a Multi-Layer Perceptron



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



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

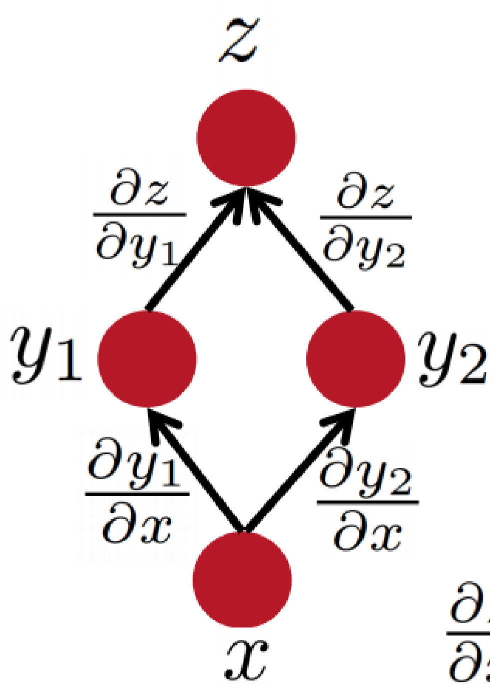
We will use these facts to derive the details of the Backpropagation algorithm.

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

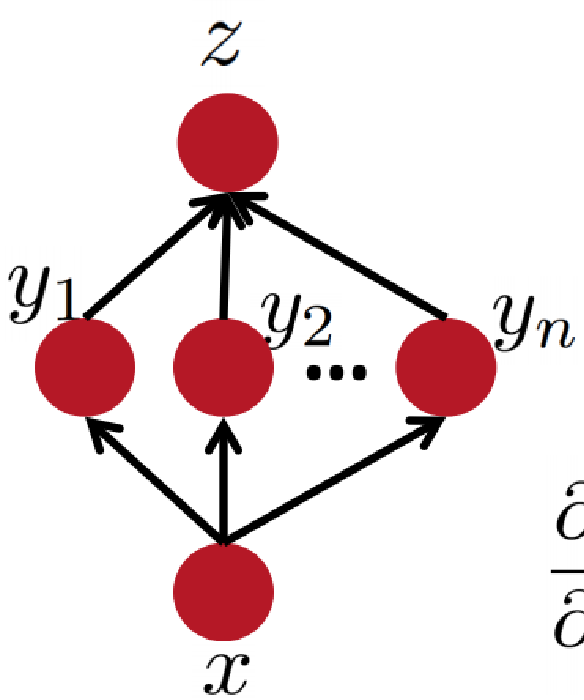


$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

Slide Credit: Richard Socher

Some facts from real analysis

- Multiple path chain rule: general



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

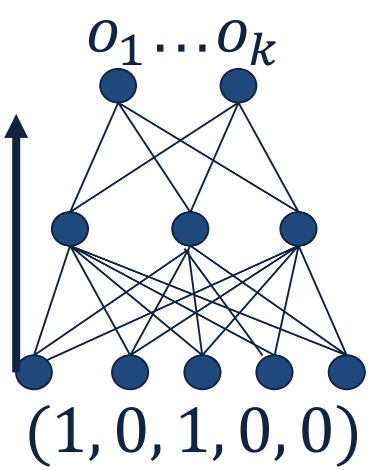
Slide Credit: Richard Socher

Backpropagation Learning Rule

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

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

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



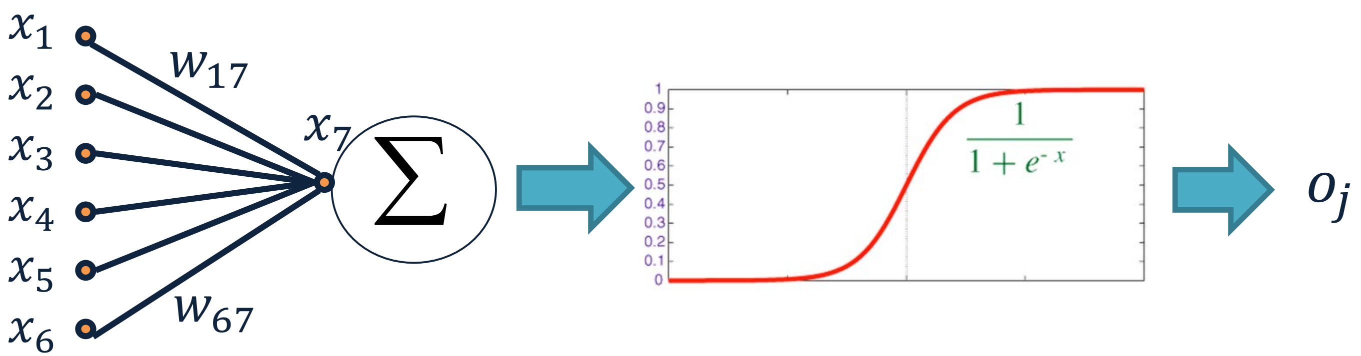
- This is used to derive the (global) learning rule which performs gradient descent in the weight space in an attempt to minimize the error function.

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

Function 1

Reminder: Model Neuron (Logistic)

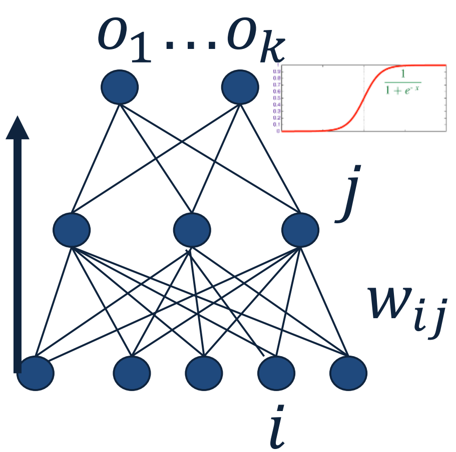
- Neuron is modeled by a unit j connected by weighted links w_{ij} to other units i.



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



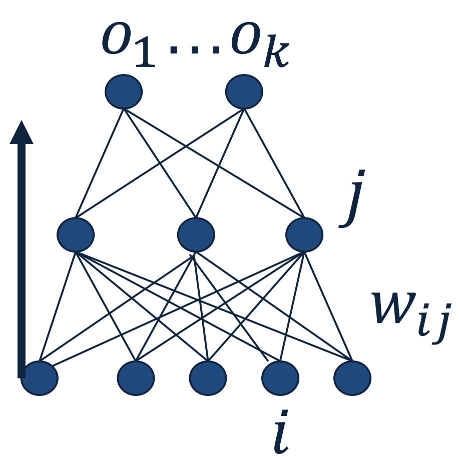
Derivation of Learning Rule

- 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

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

- Therefore:

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



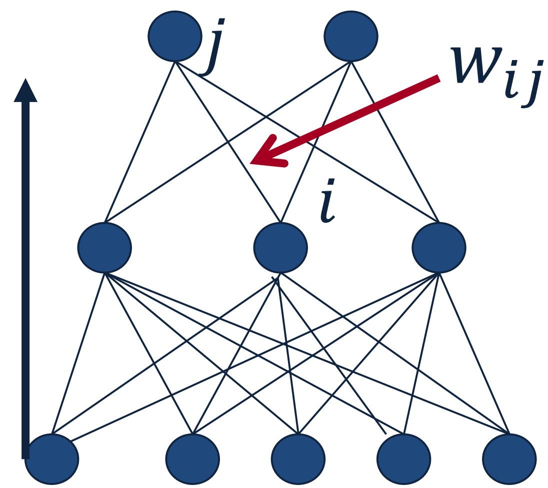
Derivation of Learning Rule (2)

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

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

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

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



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

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

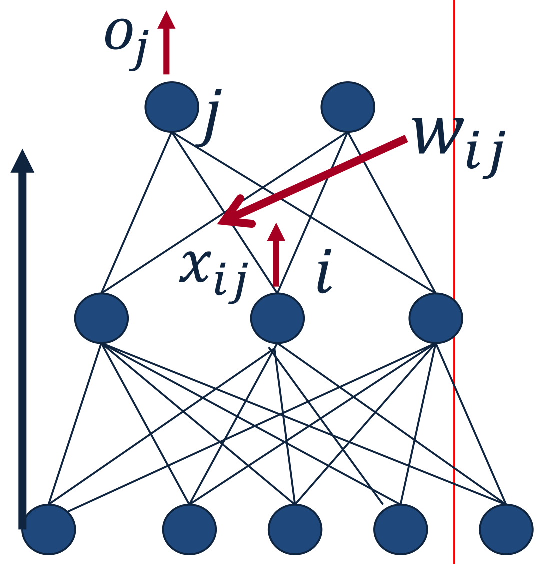
$$o_j = \frac{1}{1+\exp\{-(\mathrm{net}_j - T_j)\}}$$

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

Derivation of Learning Rule (3)

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

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



where

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

Derivation of Learning Rule (4)

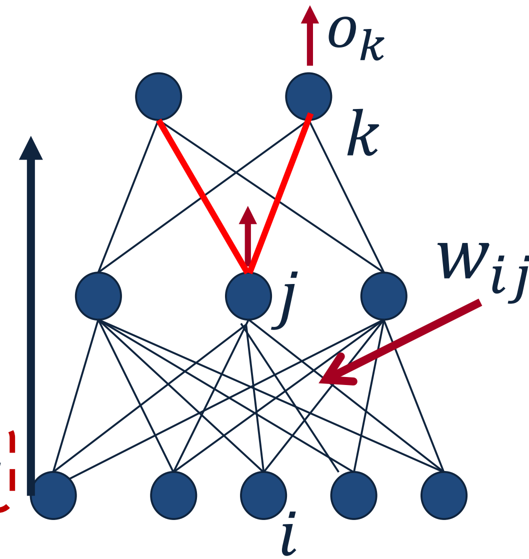
- Weights of hidden units:
  - w_{ij} Influences the output only through all the units whose direct input include j

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

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

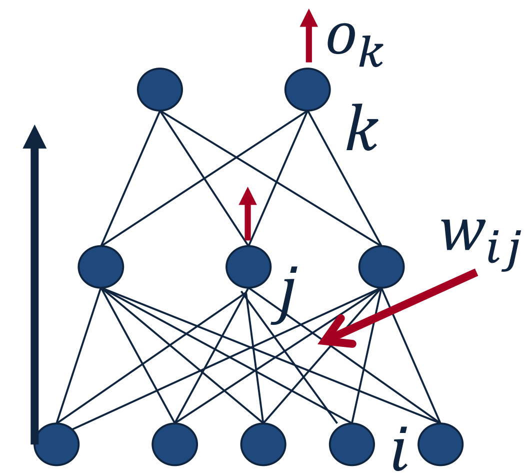
$$= \sum_{k \in downstream(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} \boxed{x_{ij}}$$

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



Derivation of Learning Rule (5)

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



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

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

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

Derivation of Learning Rule (6)

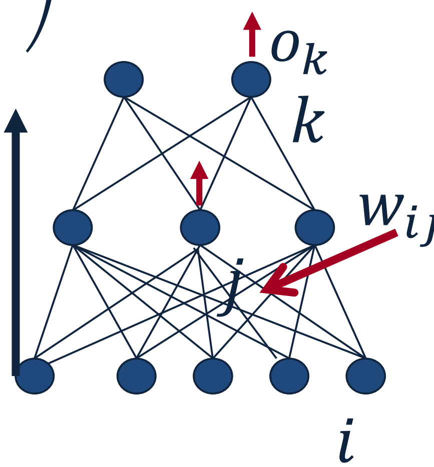
- Weights of hidden units:
  - w_{ij} is changed by:

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

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

- Where

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



- 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)

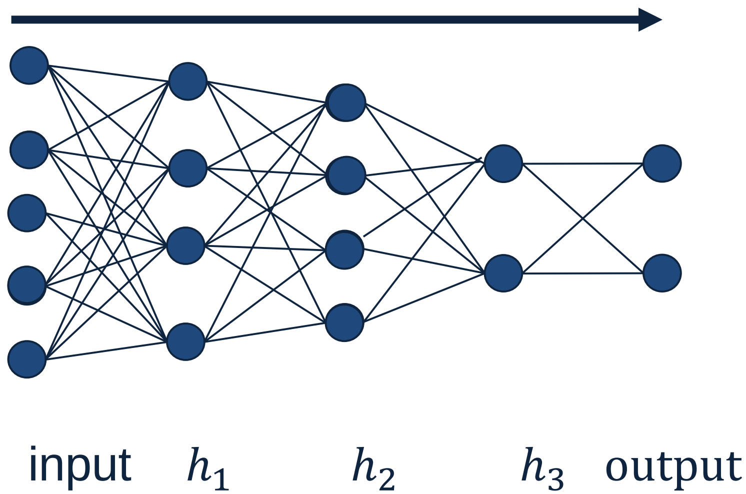
For each example in the training set do:

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

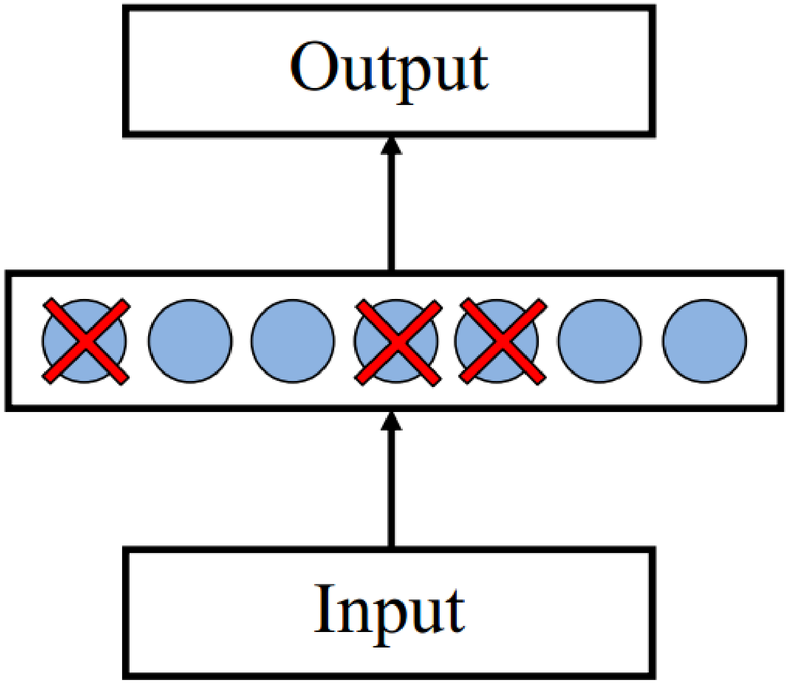
- Running too many epochs may **over-train** the network and result in over-fitting. (improved result on training, decrease in performance on test set)
- Keep an **hold-out validation** set and test accuracy after every epoch
- Maintain weights for best performing network on the validation set and return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

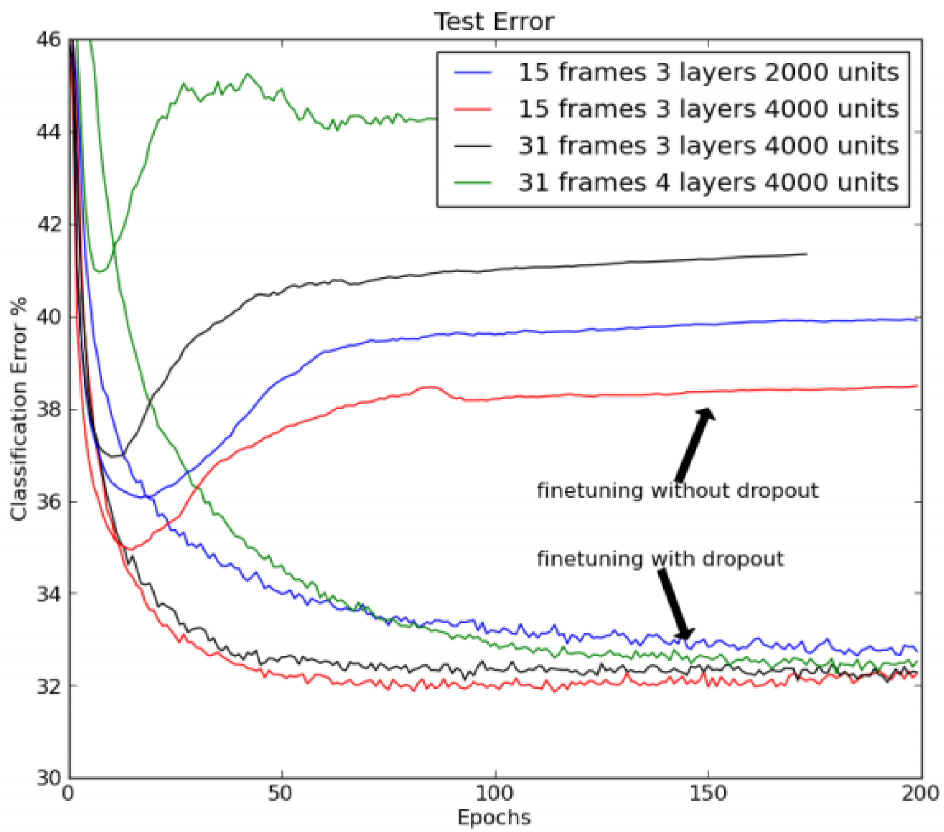
Dropout training

- Proposed by (Hinton et al, 2012)



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

Dropout training

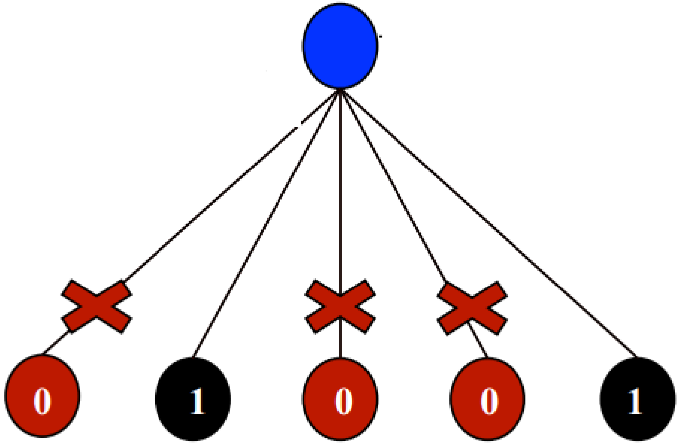


- Dropout of 50% of the hidden units and 20% of the input units (Hinton et al, 2012)

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

- Appropriate coding of inputs and outputs can make learning problem easier and improve generalization.
- Encode each binary feature as a separate input unit;

> 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 mu t per value r tion in fewer u
  - Very 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.

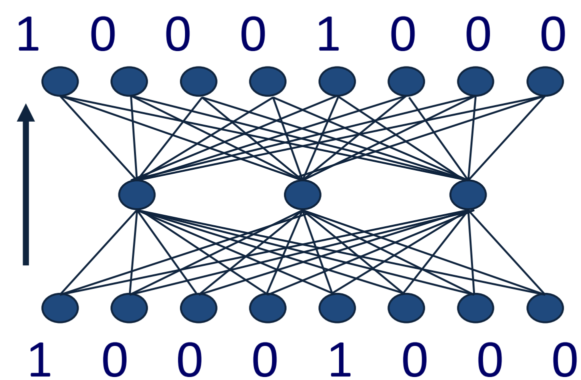
Hidden Layer Representation

- 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

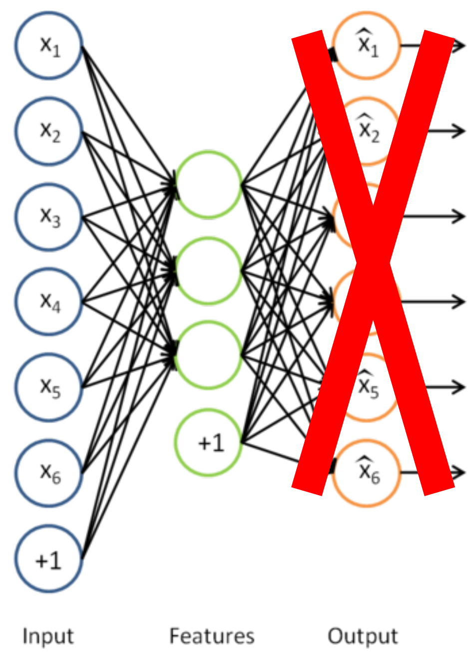
- Encoding: $\boldsymbol{y} = f(W\boldsymbol{x} + \boldsymbol{b})$
- Decoding: $\widehat{\boldsymbol{x}} = g(W'\boldsymbol{y} + \boldsymbol{b}')$
  - Goal: perfect reconstruction of input vector $\boldsymbol{x}$, by the output $\widehat{\boldsymbol{x}} = h_{\boldsymbol{\theta}}(\boldsymbol{x})$
    - 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

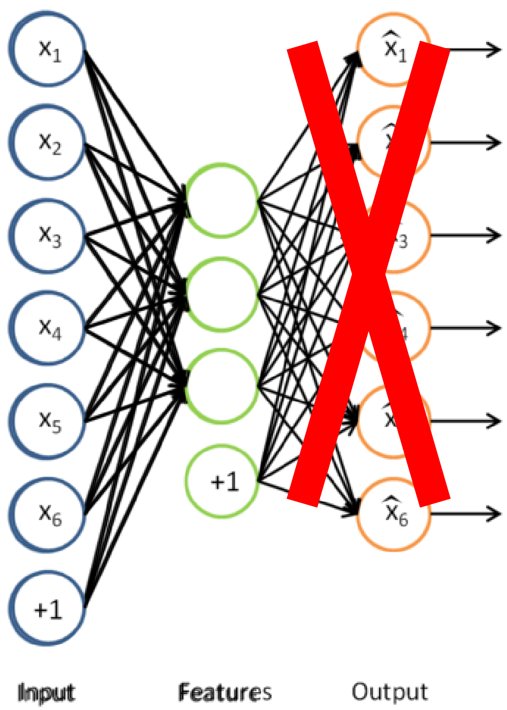
$$\min_\theta \sum_{\boldsymbol{x}} l\,(h_\theta(\boldsymbol{x}), \boldsymbol{x}) + \sum_i |w_i|$$

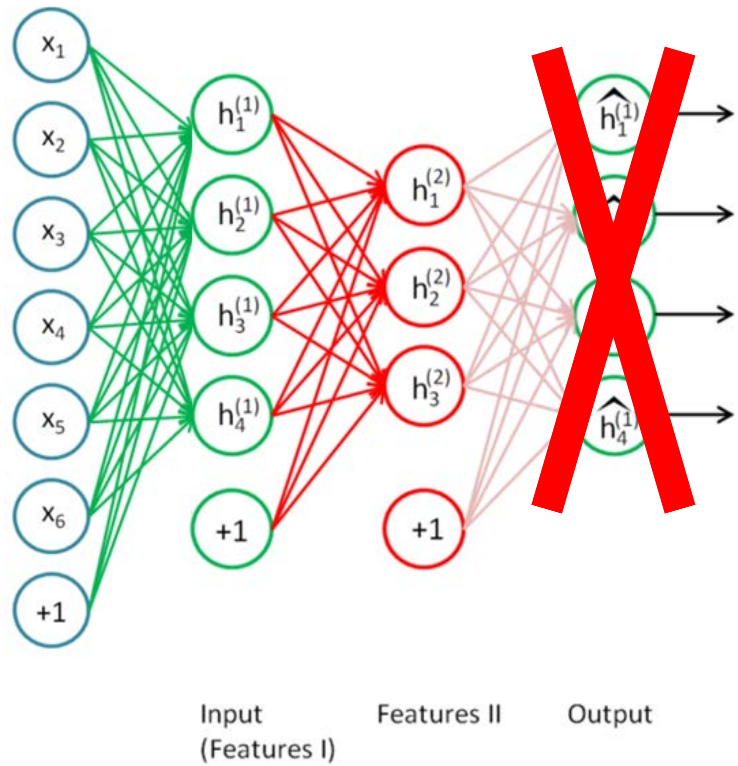
- After optimization drop the reconstruction layer and add a new layer



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

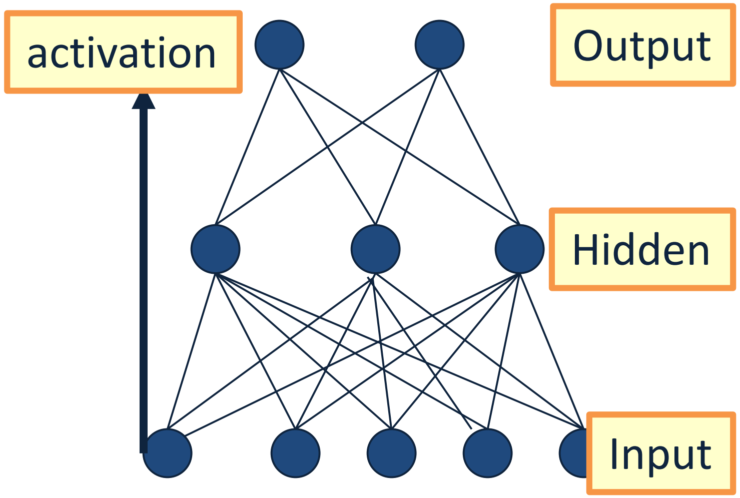
- So far what we had was purely **supervised**.
  - Initialize parameters randomly
  - Train in supervised mode typically, using backprop
  - Used in most practical systems (e.g. speech and image recognition)
- Unsupervised, layer-wise + supervised classifier on top
  - Train each layer unsupervised, 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.

NN-2

Recap: Multi-Layer Perceptrons

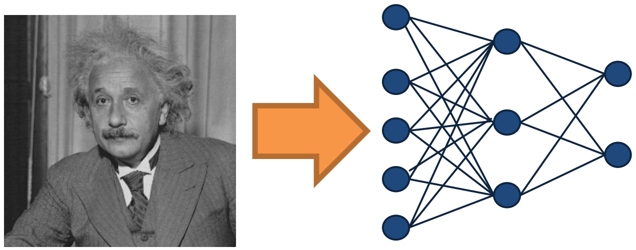
- Multi-layer network
  - A global approximator
  - Different rules for training it
- The Back-propagation
  - Forward step
  - Back propagation of errors



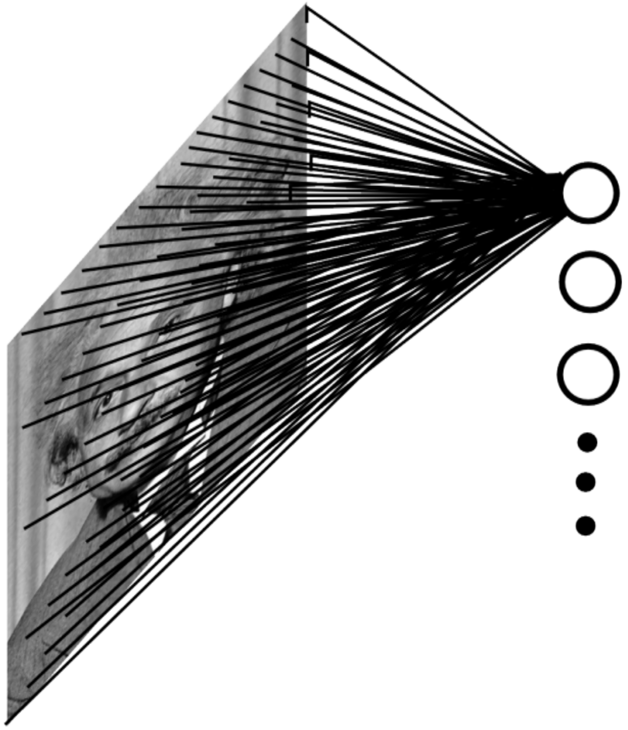
- Congrats! Now you know the hardest concept about neural networks!
- Today:
  - Convolutional Neural Networks
  - Recurrent Neural Networks

Receptive Fields

- The **receptive field** of an individual sensory neuron is the particular region of the sensory space (e.g., the body surface, or the retina) in which a stimulus will trigger the firing of that neuron.
  - In the auditory system, receptive fields can correspond to volumes in auditory space
- Designing “proper” receptive fields for the input Neurons is a significant challenge.
- Consider a task with image inputs
  - Receptive fields should give expressive features from the raw input to the system
  - How would you design the receptive fields for this problem?

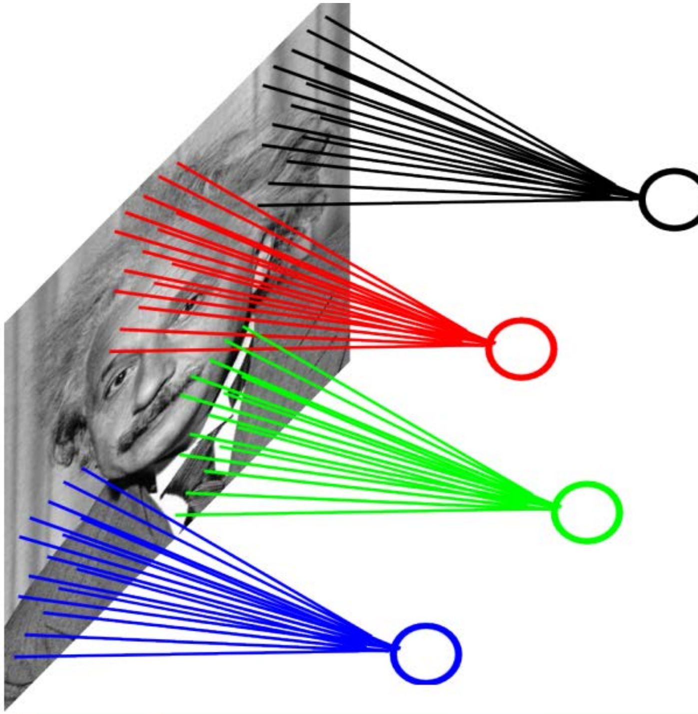


- A **fully connected layer**:
  - Example:
    - 100x100 images
    - 1000 units in the input
  - Problems:
    - 10^7 edges!
    - Spatial correlations lost!
    - Variables sized inputs.



Slide Credit: Marc'Aurelio Ranzato

- Consider a task with image inputs:
- A **locally connected layer**:
  - Example:
    - 100x100 images
    - 1000 units in the input
    - Filter size: 10x10
  - Local correlations preserved!
  - Problems:
    - 10^5 edges
    - This parameterization is good when input image is registered (e.g., face recognition).
    - Variable sized inputs, again.

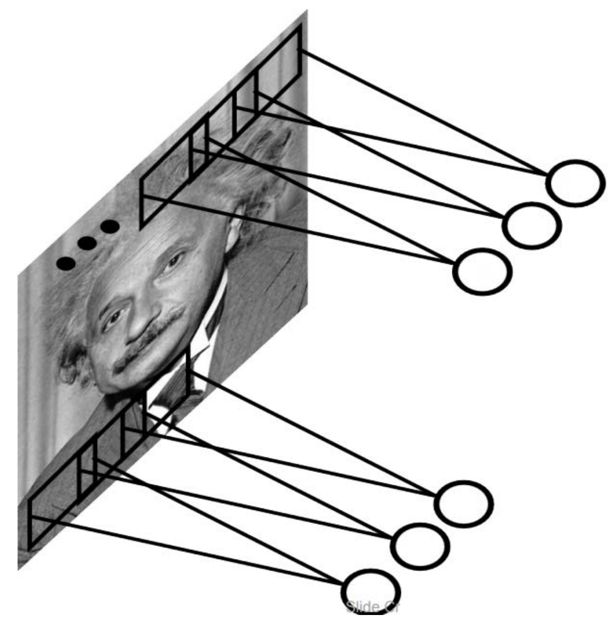


Slide Credit: Marc'Aurelio Ranzato

Convolutional Layer

- **A solution:**
  - **Filters** to capture different patterns in the input space.
    - **Share** parameters across different locations (assuming input is stationary)
    - **Convolutions** with learned filters
  - Filters will be **learned** during training.
  - The issue of variable-sized inputs will be resolved with a **pooling** layer.

So what is a convolution?



Slide Credit: Marc'Aurelio Ranzato

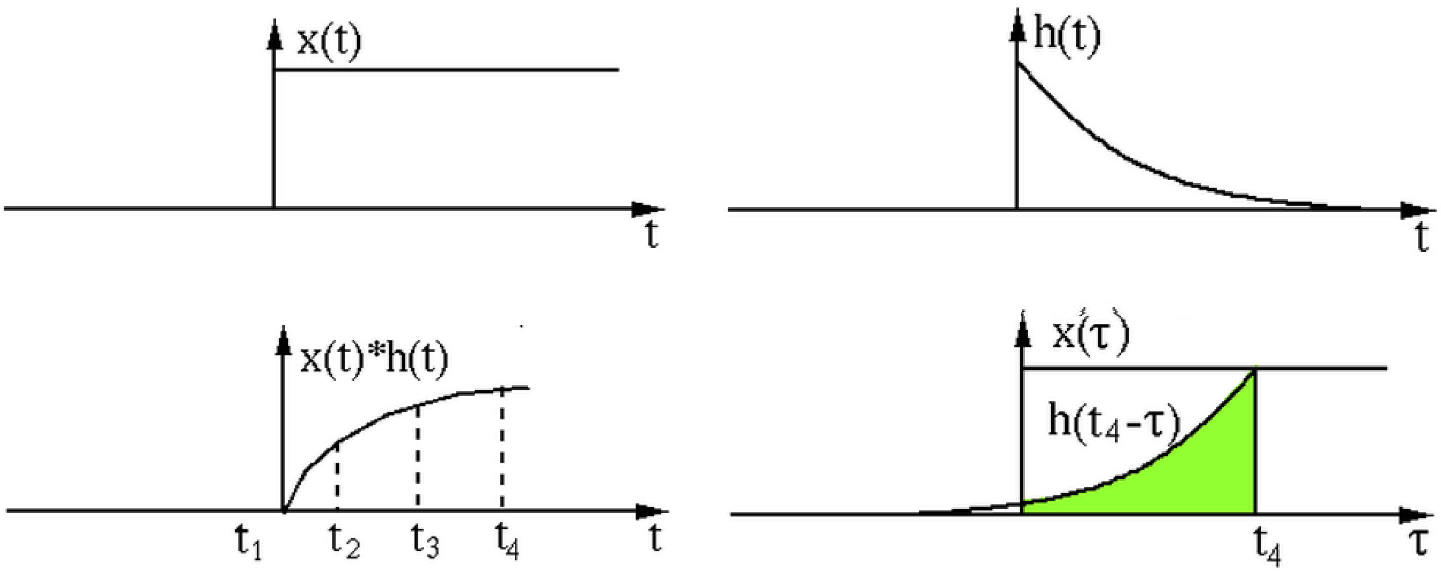
Convolution Operator

- Convolution operator: $*$
  - takes two functions and gives another function
- One dimension:

$$(x * h)(t) = \int x(\tau)h(t-\tau)d\tau$$

$$(x * h)[n] = \sum_m x[m]h[n-m]$$

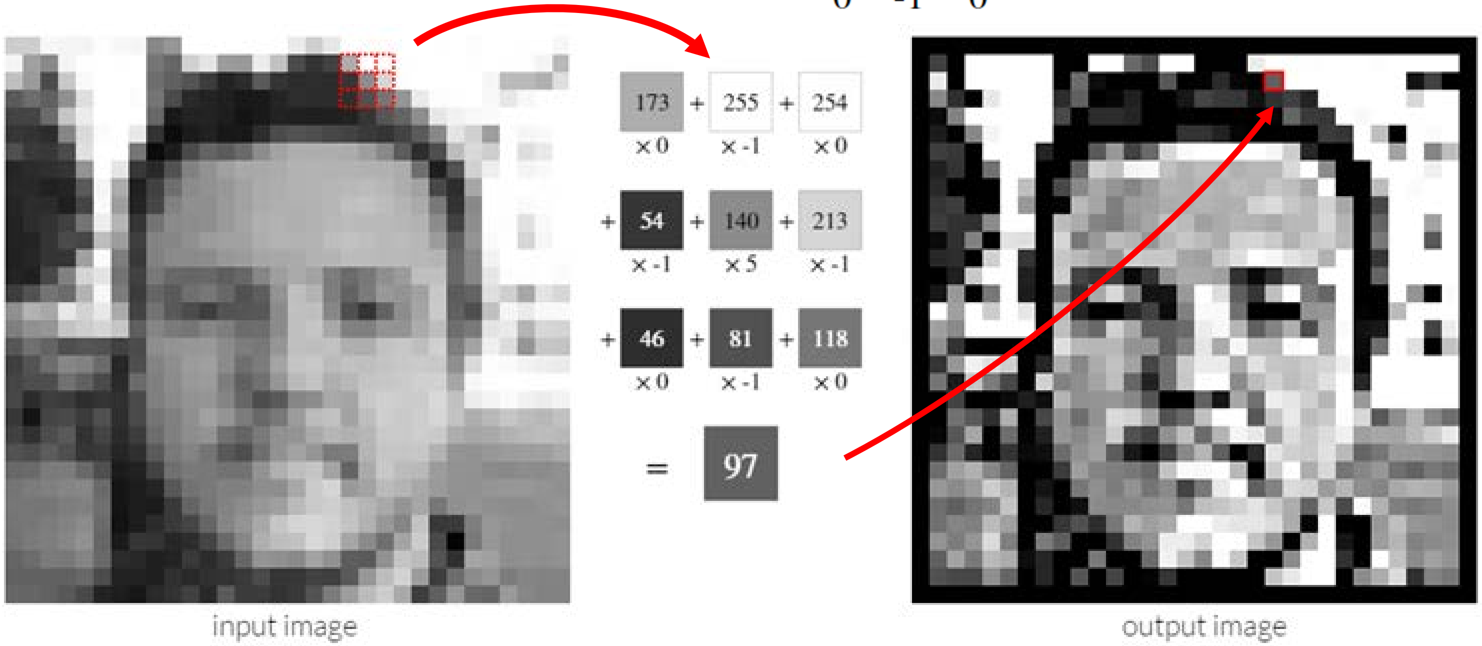
"Convolution" is very similar to "cross-correlation", except that in convolution one of the functions is flipped.



Convolution Operator (2)

- Convolution in two dimension:
  - The same idea: flip one matrix and slide it on the other matrix
  - Example: Sharpen kernel:

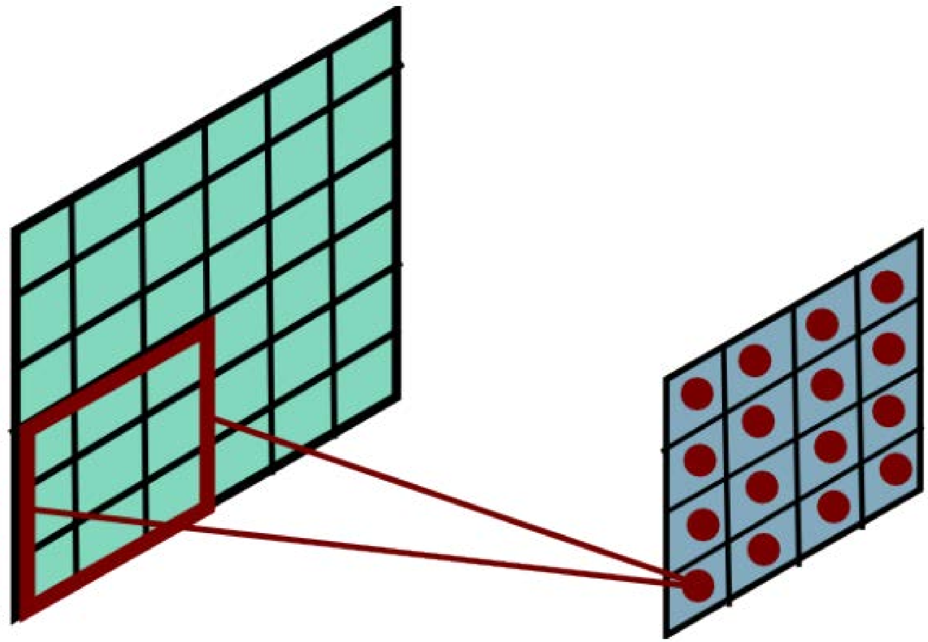
0	-1	0
-1	5	-1
0	-1	0



Try other kernels: http://setosa.io/ev/image-kernels/

Convolution Operator (3)

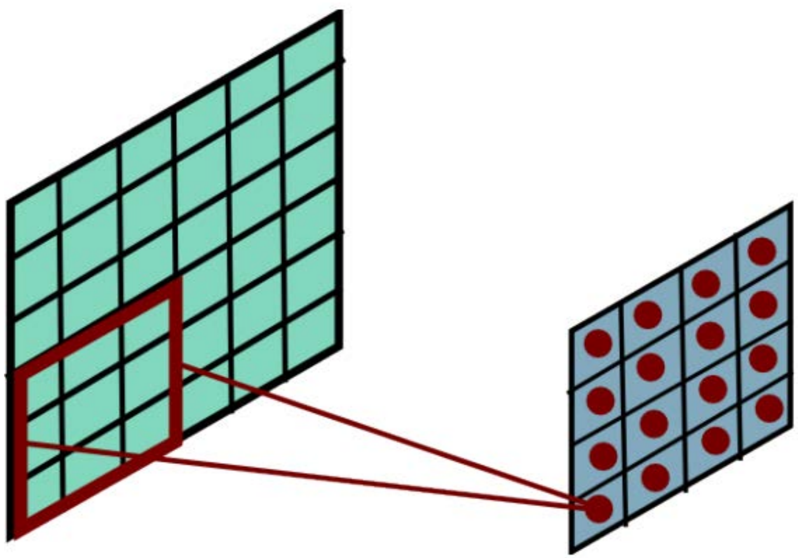
- Convolution in two dimension:
  - The same idea: flip one matrix and slide it on the other matrix



Slide Credit: Marc'Aurelio Ranza

Complexity of Convolution

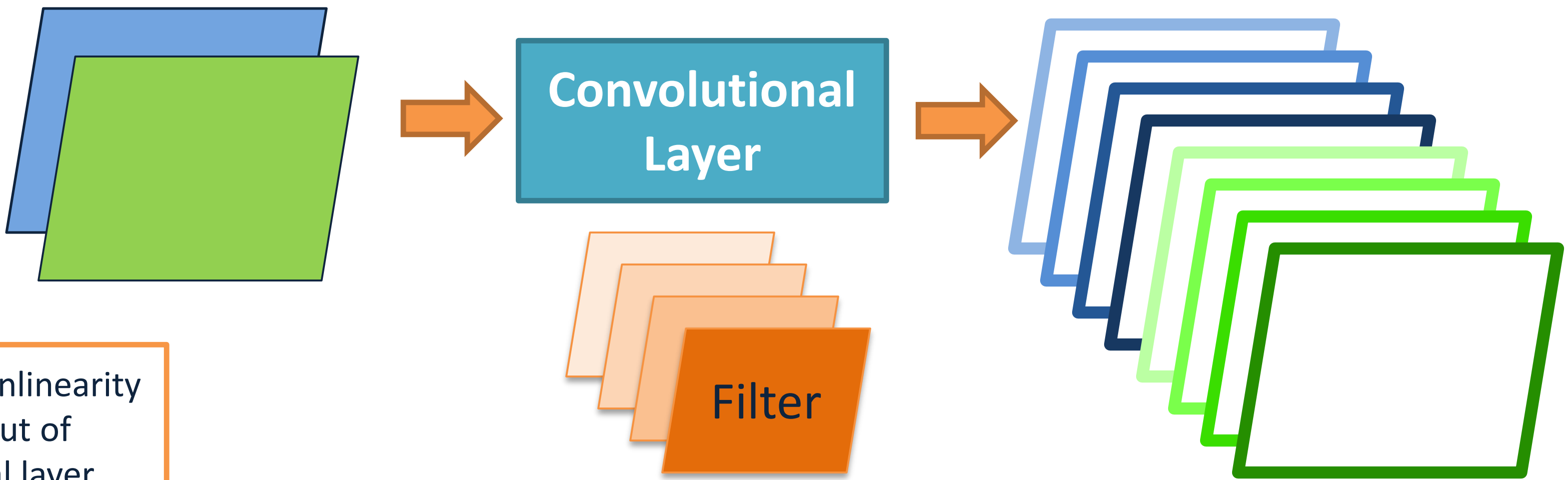
- Complexity of convolution operator is $n\log(n)$, for n inputs.
  - Uses Fast-Fourier-Transform (FFT)
- For two-dimension, each convolution takes $MN\log(MN)$ time, where the size of input is MN.



Slide Credit: Marc'Aurelio Ranzato

Convolutional Layer

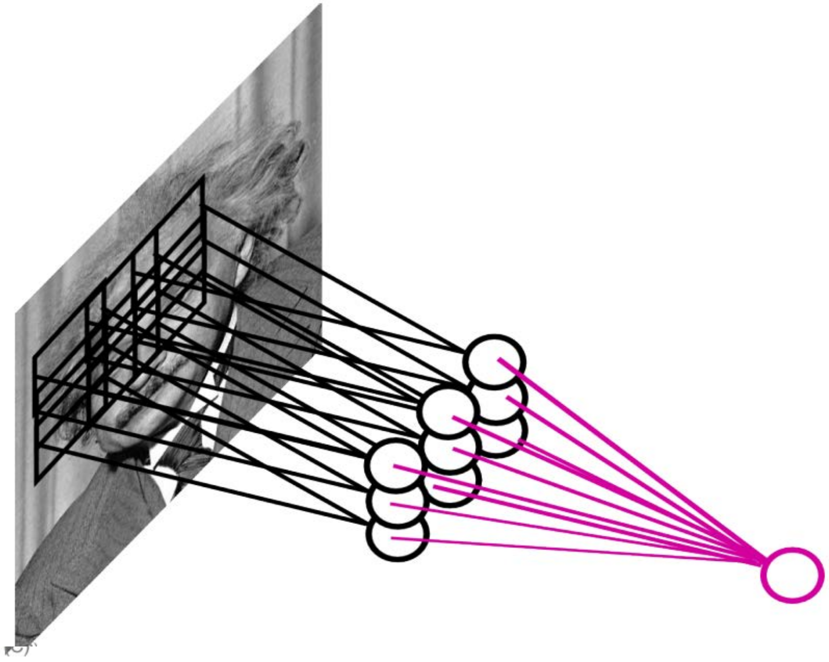
- The convolution of the **input (vector/matrix)** with weights **(vector/matrix)** results in a **response vector/matrix**.
- We can have **multiple filters** in each convolutional layer, each producing an output.
- If it is an intermediate layer, it can have **multiple inputs**!



One can add nonlinearity at the output of convolutional layer

Pooling Layer

- How to handle variable sized inputs?
  - A layer which reduces inputs of different size, to a fixed size.
  - **Pooling**



Slide Credit: Marc'Aurelio Ranzato

Pooling Layer

- How to handle variable sized inputs?
  - A layer which reduces inputs of different size, to a fixed size.
  - **Pooling**
  - Different variations
    - Max pooling

      $$h_i[n] = \max_{i \in N(n)} \tilde{h}[i]$$

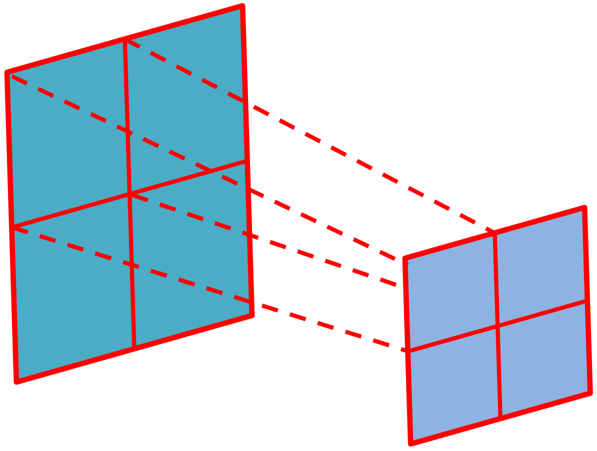
    - Average pooling

      $$h_i[n] = \frac{1}{n} \sum_{i \in N(n)} \tilde{h}[i]$$

    - L2-pooling

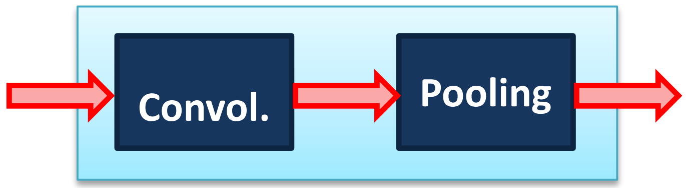
      $$h_i[n] = \frac{1}{n} \sqrt{\sum_{i \in N(n)} \tilde{h}^2[i]}$$

    - etc

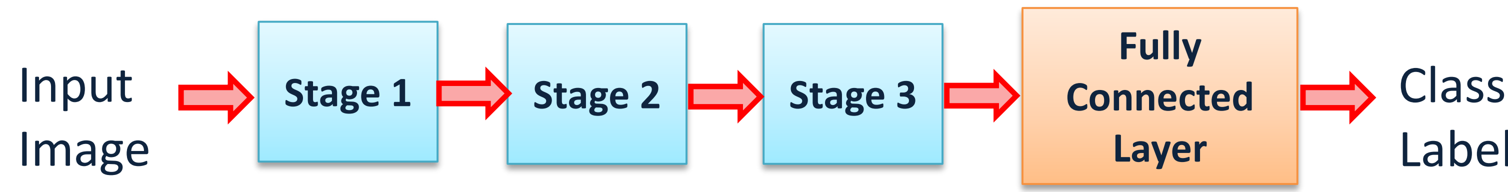


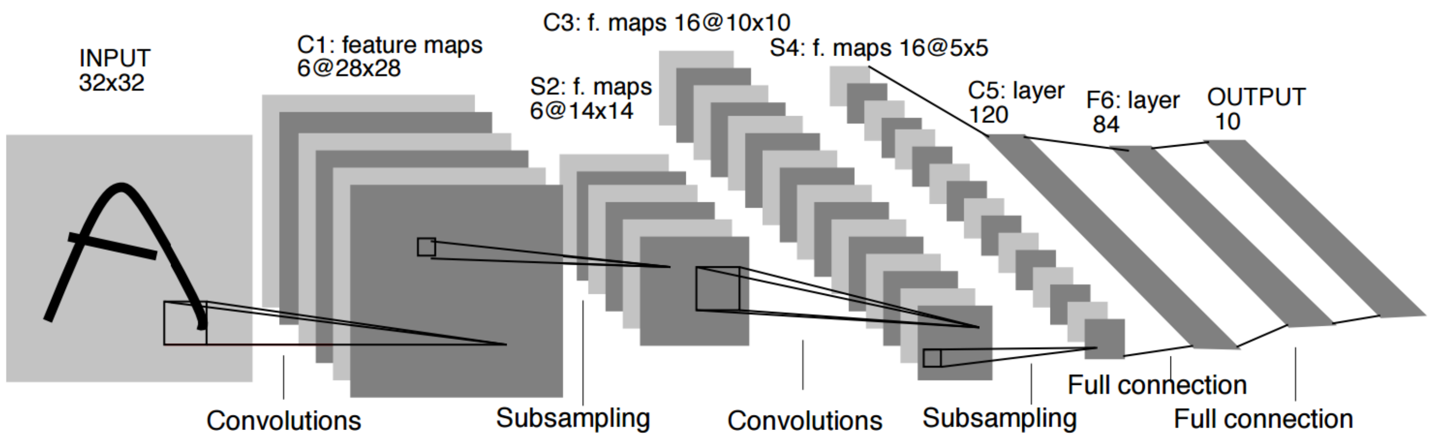
Convolutional Nets

- One stage structure:



- Whole system:

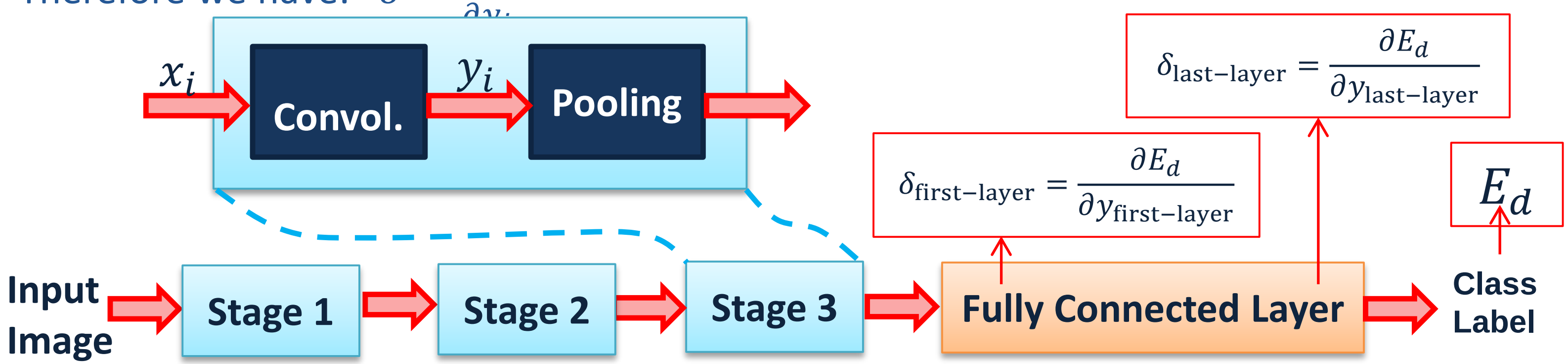




Slide Credit: Dhruv Batra

Training a ConvNet

- The same procedure from Back-propagation applies here.
  - Remember in backprop we started from the error terms in the last stage, and passed them back to the previous layers, one by one.
- Back-prop for the pooling layer:
  - Consider, for example, the case of "max" pooling.
  - This layer only routes the gradient to the input that has the highest value in the forward pass.
  - Hence, during the forward pass of a pooling layer it is common to keep track of the index of the max activation (sometimes also called *the switches*) so that gradient routing is efficient during backpropagation.
  - Therefore we have: $\delta = \frac{\partial E_d}{\partial y_i}$



Training a ConvNet

We derive the update rules for a 1D convolution, but the idea is the same for bigger dimensions.

- Back-prop for the convolutional layer:

$$\tilde{y} = w * x \quad \Leftrightarrow \quad \tilde{y}_i = \sum_{a=0}^{m-1} w_a\, x_{i-a} = \sum_{a=0}^{m-1} w_{i-a}\, x_a$$

The convolution

$$y = f(\tilde{y}) \quad \Leftrightarrow \quad y_i = f(\tilde{y}_i) \quad \forall i$$

A differentiable nonlinearity

$$\frac{\partial E_d}{\partial w_a} = \sum_{i=0}^{m-1} \frac{\partial E_d}{\partial \tilde{y}_i} \boxed{\frac{\partial \tilde{y}_i}{\partial w_a}} = \sum_{i=0}^{m-1} \boxed{\frac{\partial E_d}{\partial \tilde{y}_i}} x_{i-a}$$

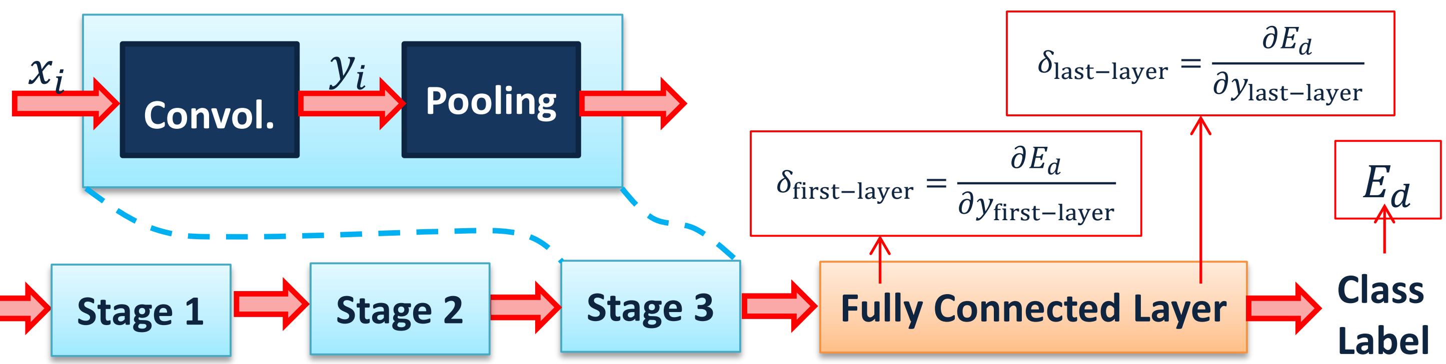
$$\frac{\partial E_d}{\partial \tilde{y}_i} = \frac{\partial E_d}{\partial y_i} \boxed{\frac{\partial y_i}{\partial \tilde{y}_i}} = \frac{\partial E_d}{\partial y_i} f'(\tilde{y})$$

Now we have everything in this layer to update the filter

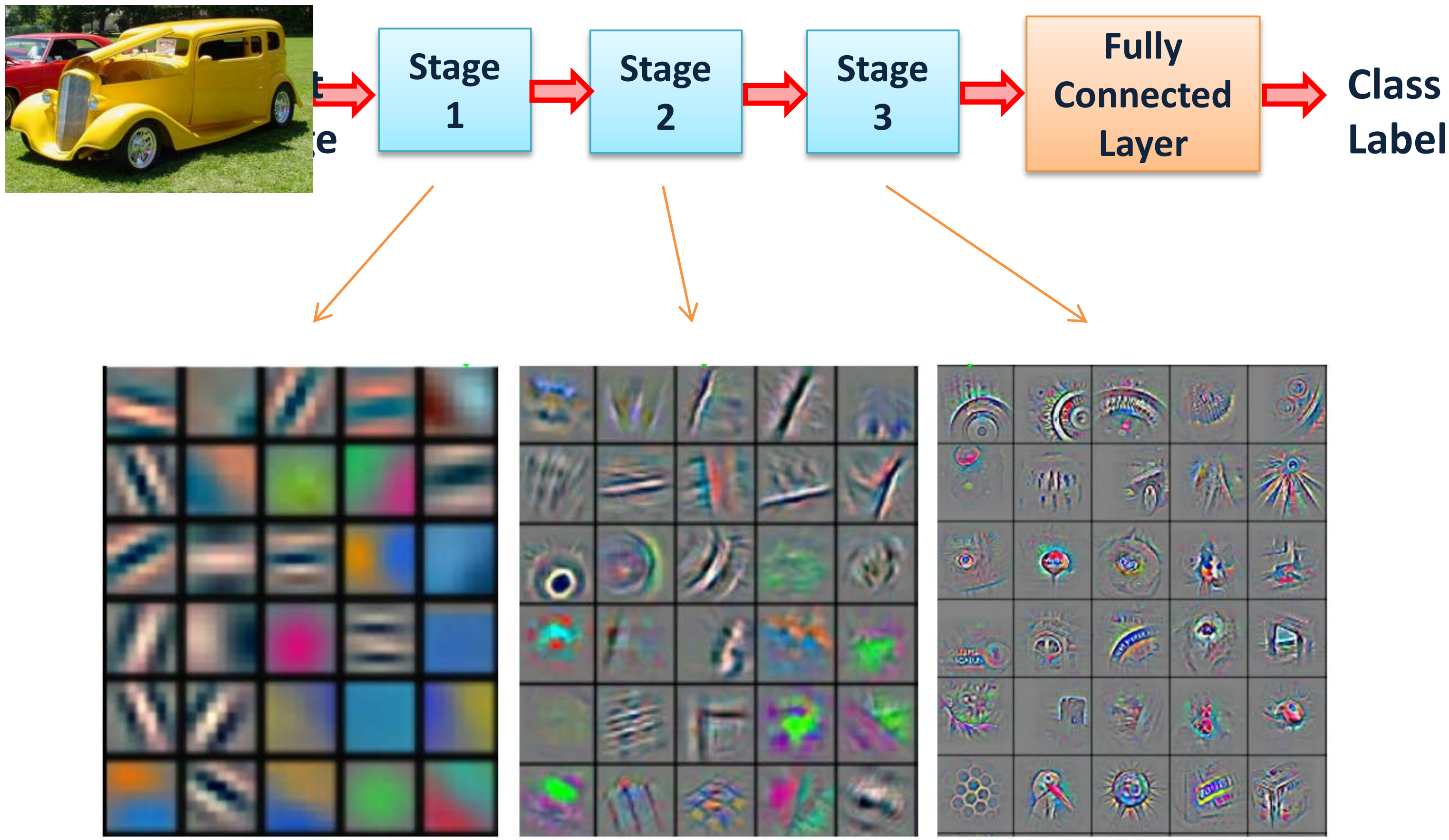
$$\delta = \frac{\partial E_d}{\partial x_a} = \sum_{i=0}^{m-1} \frac{\partial E_d}{\partial \tilde{y}_i} \boxed{\frac{\partial \tilde{y}_i}{\partial x_a}} = \sum_{i=0}^{m-1} \frac{\partial E_d}{\partial \tilde{y}_i} w_{i-a}$$

We need to pass the gradient to the previous layer

Now we can repeat this for each stage of ConvNet.



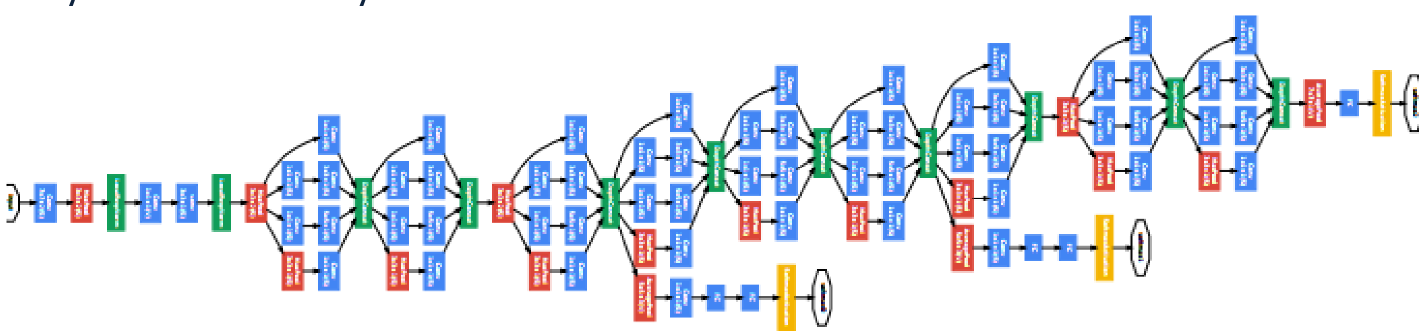
Convolutional Nets



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

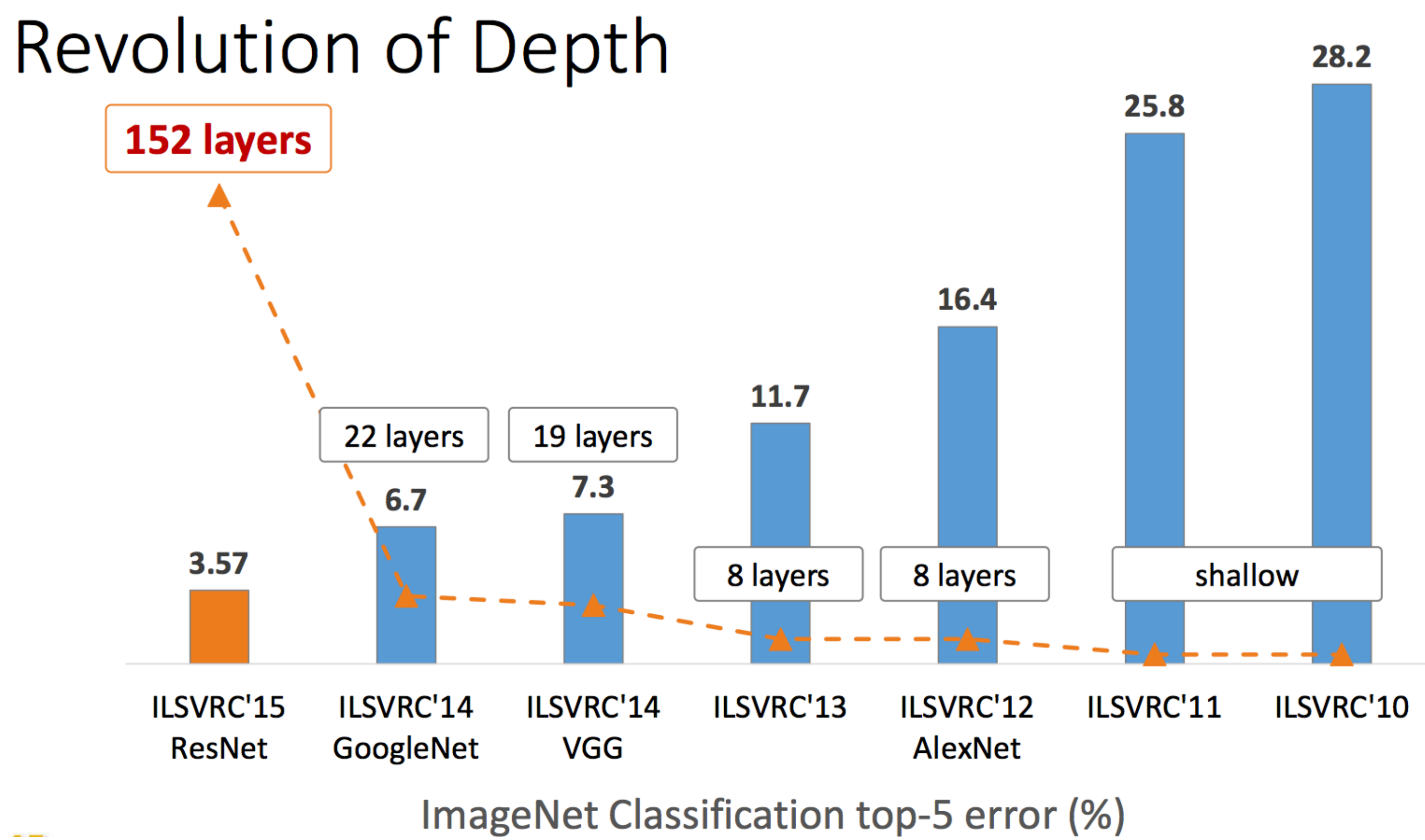
ConvNet roots

- **Fukushima, 1980s** designed network with same basic structure but did not train by backpropagation.
- The first successful applications of **Convolutional Networks** by Yann LeCun in 1990's (LeNet)
  - Was used to read zip codes, digits, etc.
- Many variants nowadays, but the core idea is the same
  - Example: a system developed in Google (GoogLeNet)
    - Compute different filters
    - Compose one big vector from all of them
    - Layer this iteratively



See more: http://arxiv.org/pdf/1409.4842v1.pdf

Depth matters



Slide from [Kaiming He 2015]

Practical Tips

- Before large scale experiments, test on a small subset of the data and check the error should go to zero.
  - Overfitting on small training
- Visualize features (feature maps need to be uncorrelated) and have high variance
- Bad training: many hidden units ignore the input and/or exhibit strong correlations.

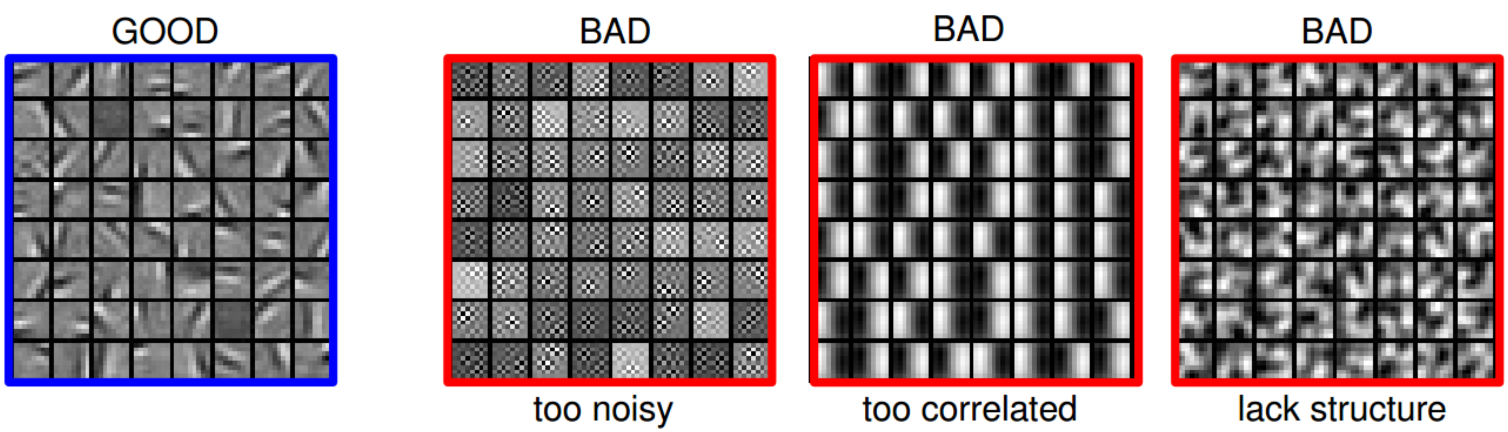


Figure Credit: Marc'Aurelio Ranzato

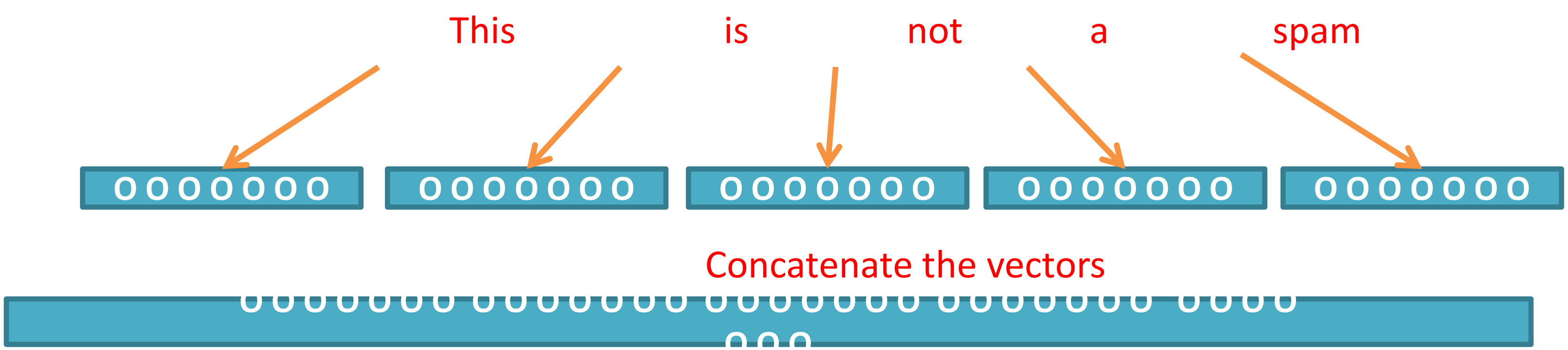
Debugging

- Training diverges:
  - Learning rate may be too large → decrease learning rate
  - BackProp is buggy → numerical gradient checking
- Loss is minimized but accuracy is low
  - Check loss function: Is it appropriate for the task you want to solve? Does it have degenerate solutions?
- NN is underperforming / under-fitting
  - Compute number of parameters → if too small, make network larger
- NN is too slow
  - Compute number of parameters → Use distributed framework, use GPU, make network smaller

Many of these points apply to many machine learning models, no just neural networks.

CNN for vector inputs

- Let's study another variant of CNN for language
  - Example: sentence classification (say spam or not spam)
- First step: represent each word with a vector in $\mathbb{R}^d$

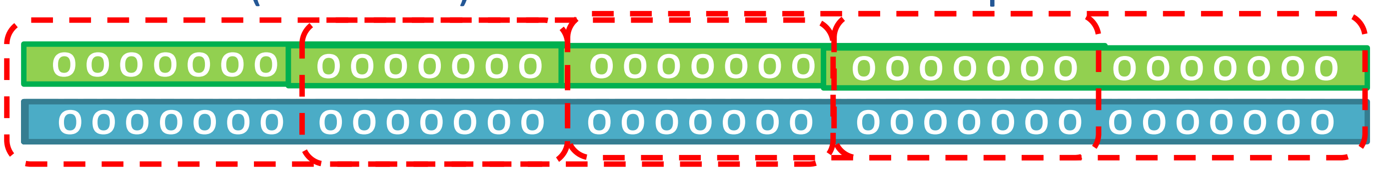


- Now we can assume that the input to the system is a vector $\mathbb{R}^{dl}$
  - Where the input sentence has length l ($l = 5$ in our example)
  - Each word vector's length d ($d = 7$ in our example)

Convolutional Layer on vectors

- Think about a single convolutional layer
  - A bunch of **vector** filters
    - Each defined in $\mathbb{R}^{dh}$
      - Where h is the number of the words the filter covers
      - Size of the word vector d

  - Find its (modified) convolution with the input vector



$c_1 = f(w.x_{1:h})$ $c_2 = f(w.x_{h+1:2h})$ $c_3 = f(w.x_{2h+1:3h})$ $c_4 = f(w.x_{3h+1:4h})$

  - Result of the convolution with the filter

$$c = [c_1, \ldots, c_{n-h+1}]$$

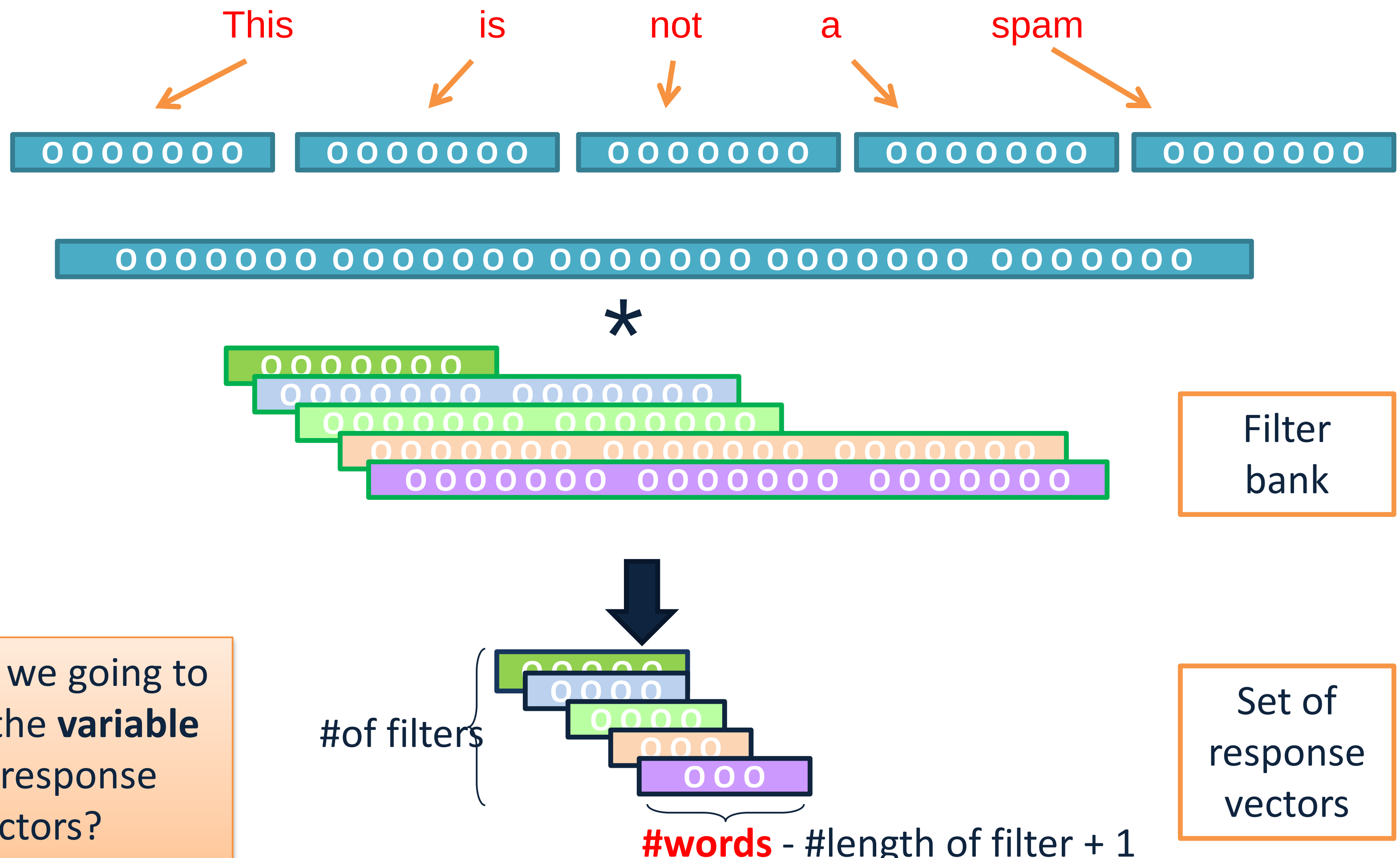
  - Convolution with a filter that spans 2 words, is operating on all of the bi-grams (vectors of two consecutive word, concatenated): “this is”, “is not”, “not a”, “a spam”.
  - Regardless of whether it is grammatical (not appealing linguistically)

Convolutional Layer on vectors

Get word vectors for each words

Concatenate vectors

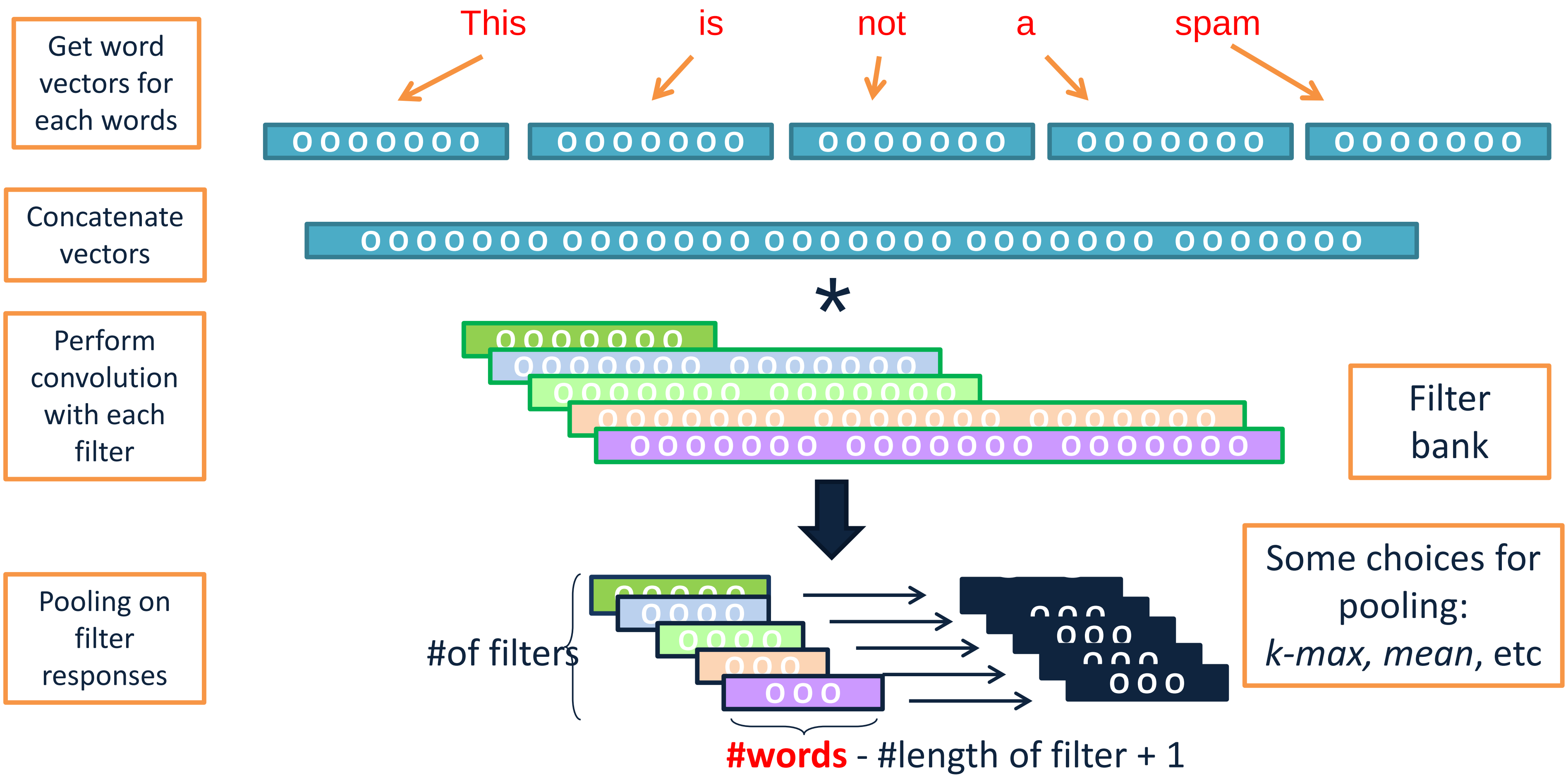
Perform convolution with each filter



How are we going to handle the **variable sized** response vectors?

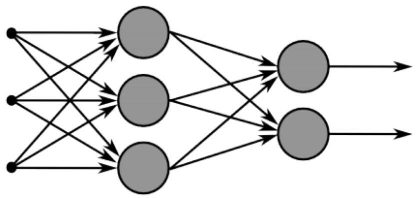
Pooling!

Convolutional Layer on vectors



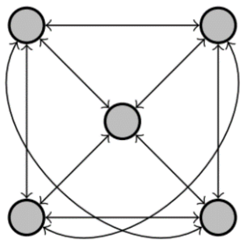
- Now we can pass the fixed-sized vector to a logistic unit (softmax), or give it to multi-layer network (last session)

Recurrent Neural Networks



- Multi-layer feed-forward NN: **DAG**
  - Just computes a fixed sequence of

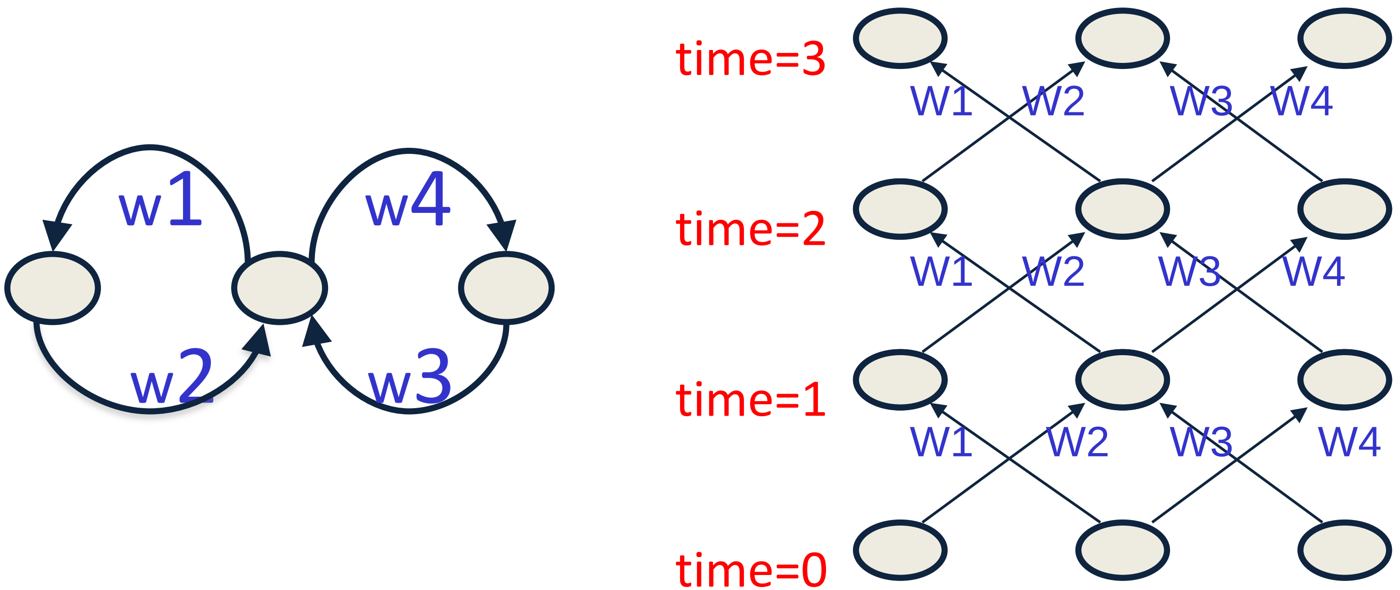
  non-linear learned transformations to convert an input patter into an output pattern



- Recurrent Neural Network: **Digraph**
  - Has cycles.
  - Cycle can act as a memory;
  - The hidden state of a recurrent net can carry along information about a “potentially” unbounded number of previous inputs.
  - They can model sequential data in a much more natural way.

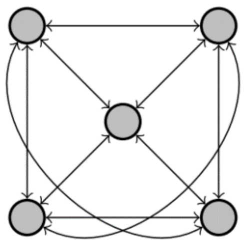
Equivalence between RNN and Feed-forward NN

- Assume that there is a time delay of 1 in using each connection.
- The recurrent net is just a layered net that keeps reusing the same weights.



Slide Credit: Geoff Hinton

Recurrent Neural Networks



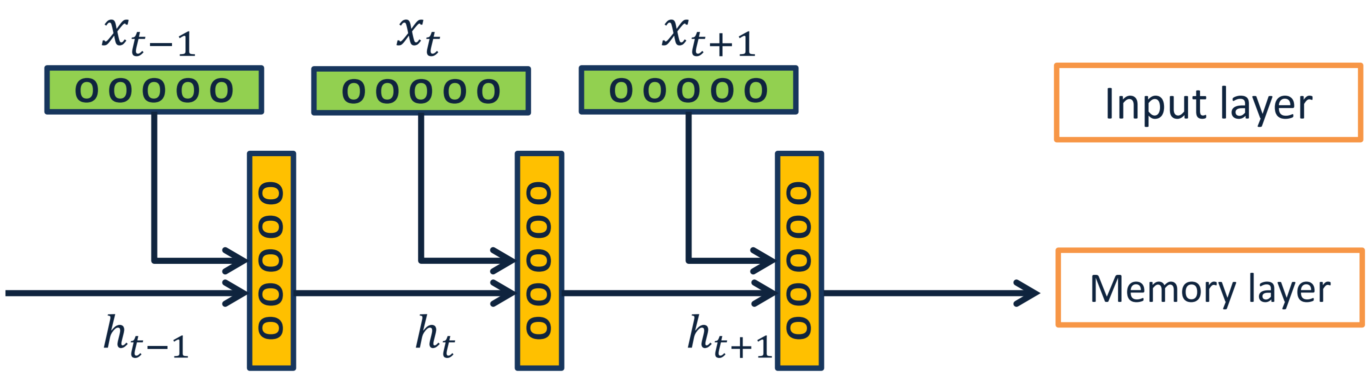
- Training a general RNN's can be hard
  - Here we will focus on a **special family of RNN's**
- Prediction on chain-like input:
  - Example: POS tagging words of a sentence

$X =$	This	is	a	sample	sentence
$Y =$	DT	VBZ	DT	NN	NN

  - Issues :
    - Structure in the output: There is connections between labels
    - Interdependence between elements of the inputs: The final decision is based on an intricate interdependence of the words on each other.
    - Variable size inputs: e.g. sentences differ in size
- How would you go about solving this task?

Recurrent Neural Networks

- A chain RNN:
  - Has a chain-like structure
  - Each input is replaced with its vector representation x_t
  - Hidden (memory) unit h_t contain information about previous inputs and previous hidden units h_{t-1}, h_{t-2}, etc
    - Computed from the past memory and current word. It summarizes the sentence up to that time.

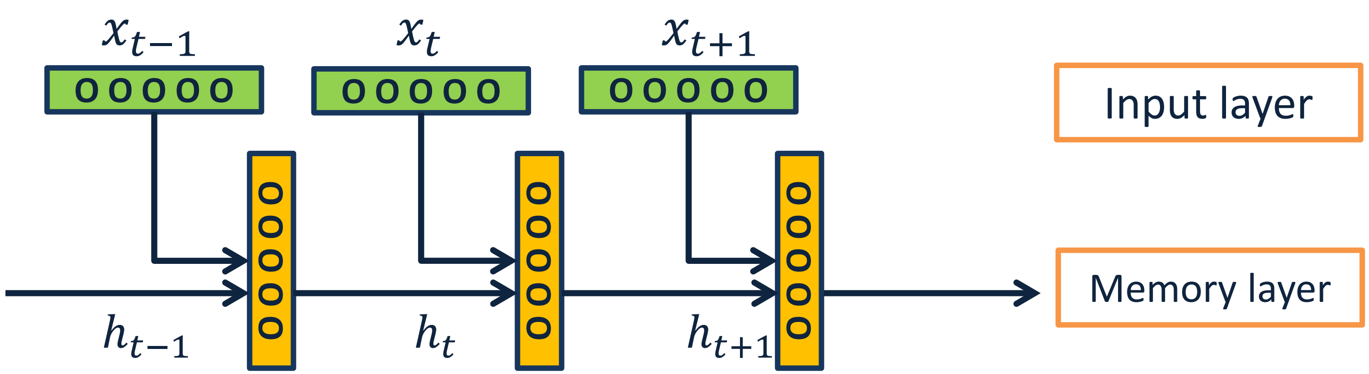


Recurrent Neural Networks

- A popular way of formalizing it:

$$h_t = f(W_h h_{t-1} + W_i x_t)$$

  - Where f is a nonlinear, differentiable (why?) function.
- Outputs?
  - Many options; depending on problem and computational resource



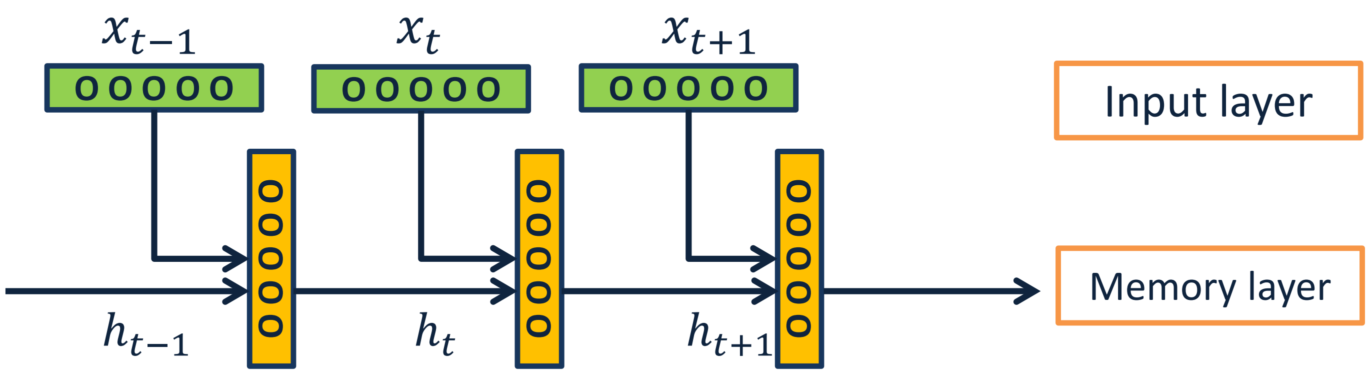
Recurrent Neural Networks

- Prediction for x_t, with h_t
- Prediction for x_t, with $h_t, \ldots, h_{t-\tau}$
- Prediction for the whole chain

$$y_t = \text{softmax}(W_o h_t)$$

$$y_t = \text{softmax}\left(\sum_{i=0}^{\tau} \alpha^i W_o \; {}^{-i} h_{t-i}\right)$$

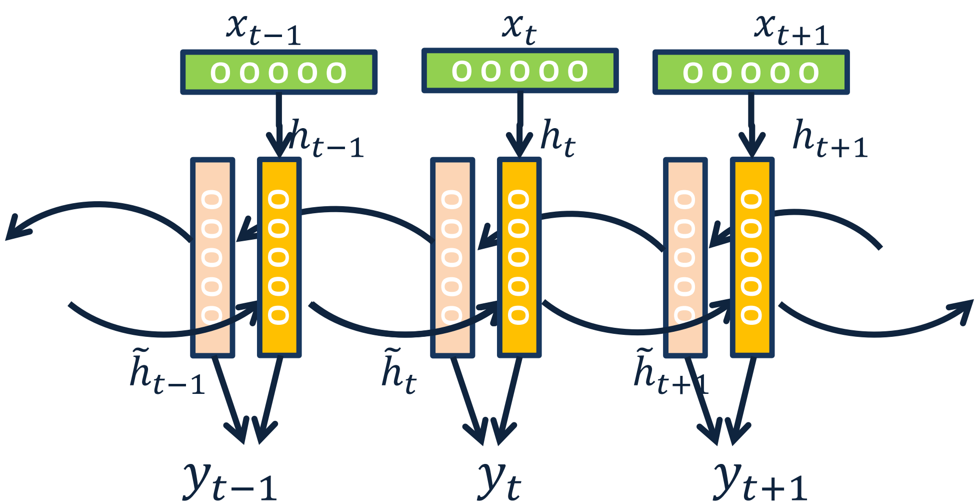
$$y_T = \text{softmax}(W_o h_T)$$



- Some inherent issues with RNNs:
  - Recurrent neural nets cannot capture phrases without prefix context
  - They often capture too much of last words in final vector

Bi-directional RNN

- One of the issues with RNN:
  - Hidden variables capture only one side context
- A bi-directional structure



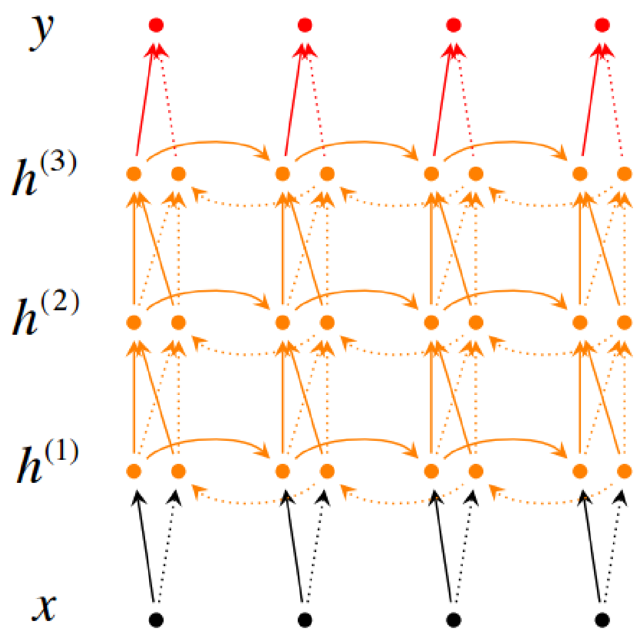
$$h_t = f(W_h h_{t-1} + W_i x_t)$$

$$\tilde{h}_t = f(\widetilde{W}_h \tilde{h}_{t+1} + \widetilde{W}_i x_t)$$

$$y_t = \text{softmax}\left(W_o h_t + \widetilde{W}_o \tilde{h}_t\right)$$

Stack of bi-directional networks

- Use the same idea and make your model further complicated:



Training RNNs

- How to train such model?
  - Generalize the same ideas from back-propagation
- Total output error: $E\left(\vec{y},\vec{t}\right) = \sum_{t=1}^{T} E_t(y_t, t_t\,)$

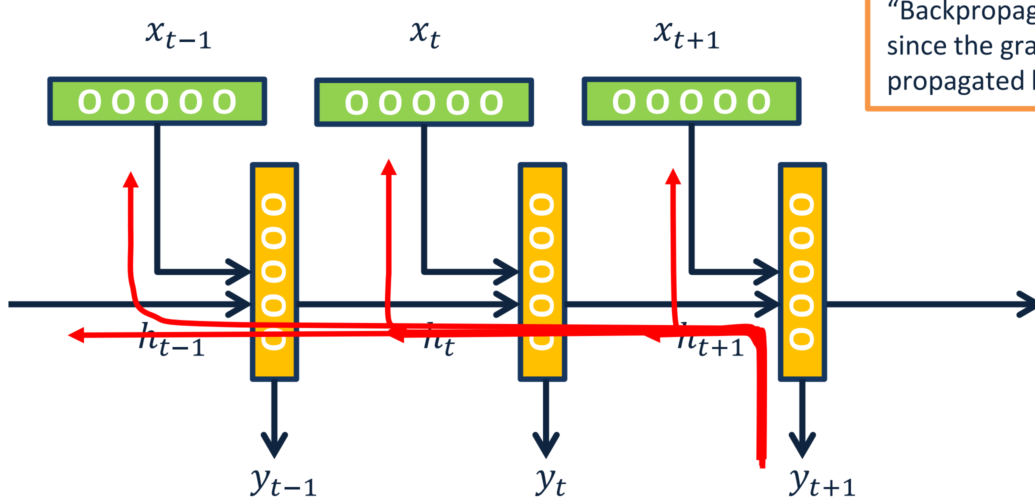
Parameters? W_o, W_i, W_h + vectors for input

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W}$$

$$\frac{\partial E_t}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \boxed{\frac{\partial h_t}{\partial h_{t-k}}} \frac{\partial h_{t-k}}{\partial W}$$

Reminder:
$y_t = \text{softmax}(W_o h_t\,)$
$h_t = f(W_h h_{t-1} + W_i x_t)$

This sometimes is called "Backpropagation Through Time", since the gradients are propagated back through time.



Recurrent Neural Network

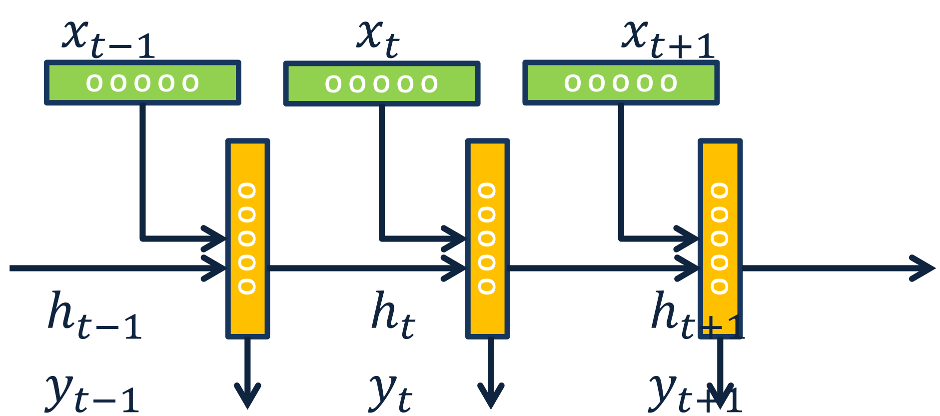
$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\boxed{\frac{\partial h_t}{\partial h_{t-k}}}\frac{\partial h_{t-k}}{\partial W}$$

Reminder:
$y_t = \text{softmax}(W_o h_t)$
$h_t = f(W_h h_{t-1} + W_i x_t)$

$$\frac{\partial h_t}{\partial h_{t-1}} = W_h \text{diag}[f'(W_h h_{t-1} + W_i x_t)]$$

$$\text{diag}[a_1, \dots, a_n] = \begin{bmatrix} a_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & a_n \end{bmatrix}$$

$$\frac{\partial h_t}{\partial h_{t-k}} = \prod_{j=t-k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=t-k+1}^{t} W_h \text{diag}[f'(W_h h_{t-1} + W_i x_t)]$$



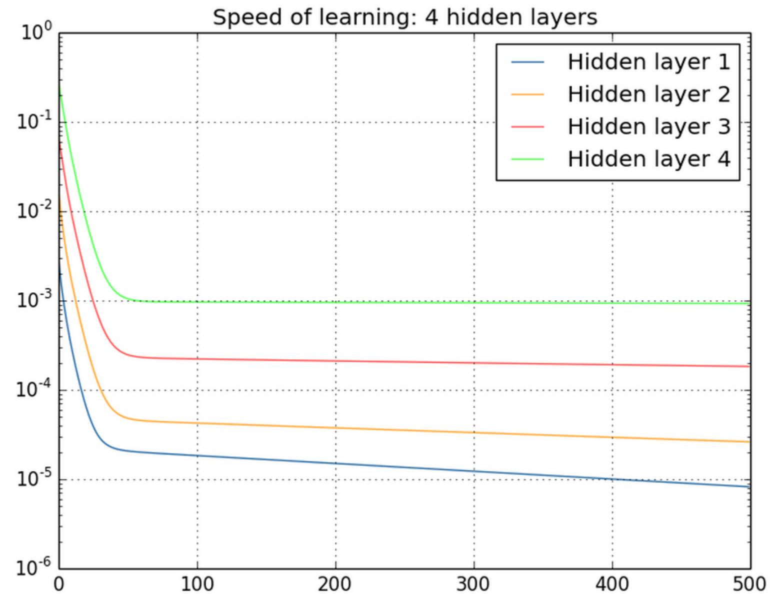
Vanishing/exploding gradients

$$\frac{\partial h_t}{\partial h_{t-k}} = \prod_{j=t-k+1}^{t} W_h \operatorname{diag}[f'(W_h h_{t-1} + W_i x_t)]$$

$$\frac{\partial h_t}{\partial h_k} \le \prod_{j=t-k+1}^{t} \|W_h\| \|\operatorname{diag}[f'(W_h h_{t-1} + W_i x_t)]\| \le \prod_{j=t-k+1}^{t} \alpha\beta = (\alpha\beta)^k$$

Gradient can become very **small or very large quickly**, and the locality assumption of gradient descent breaks down (Vanishing gradient) [Bengio et al 1994]

- Vanishing gradients are quite prevalent and a serious issue.
- A real example
  - Training a feed-forward network
  - y-axis: sum of the gradient norms
  - Earlier layers have exponentially smaller sum of gradient norms
  - This will make training earlier layers much slower.



Vanishing/exploding gradients

- In an RNN trained on long sequences (*e.g.* 100 time steps) the gradients can easily explode or vanish.
  - So RNNs have difficulty dealing with long-range dependencies.
- Many methods proposed for reduce the effect of vanishing gradients; although it is still a problem
  - Introduce shorter path between long connections
  - Abandon stochastic gradient descent in favor of a much more sophisticated Hessian-Free (HF) optimization
  - Add fancier modules that are robust to handling long memory; e.g. Long Short Term Memory (LSTM)
- One trick to handle the exploding-gradients:
  - Clip gradients with bigger sizes:

Defnne $g = \frac{\partial E}{\partial W}$

If $\|g\| \geq threshold$ then

$$g \leftarrow \frac{threshold}{\|g\|} g$$