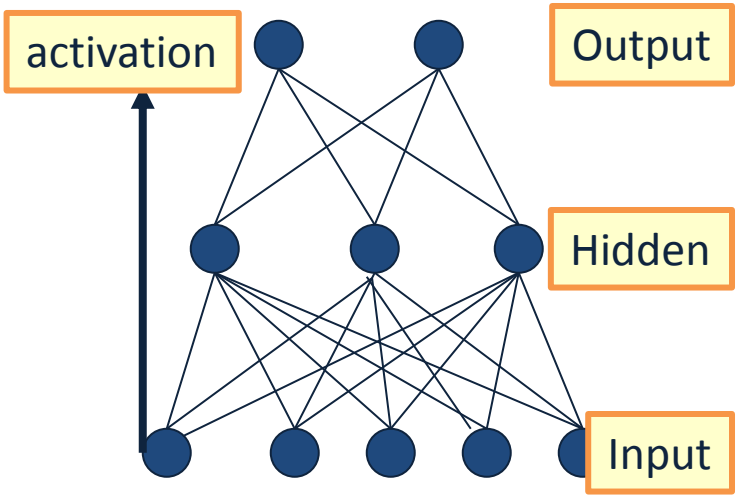
Administration

Questions

- Hw4 is out
  - Please start working on it as soon as possible
  - Come to sections with questions
- Deadline for project proposals is close

Recap: Multi-Layer Perceptrons

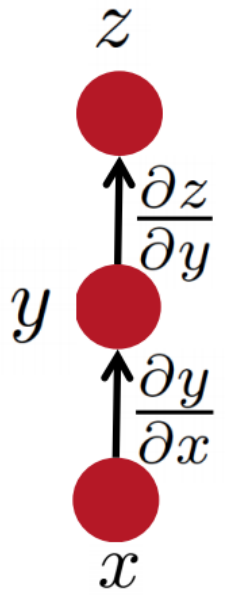
- Multi-layer network
  - Different rules for training it
- Three layer network
  - A global approximator
- The Back-propagation
  - Forward step
  - Back propagation of errors



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

Some facts from real analysis

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



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

Reminder

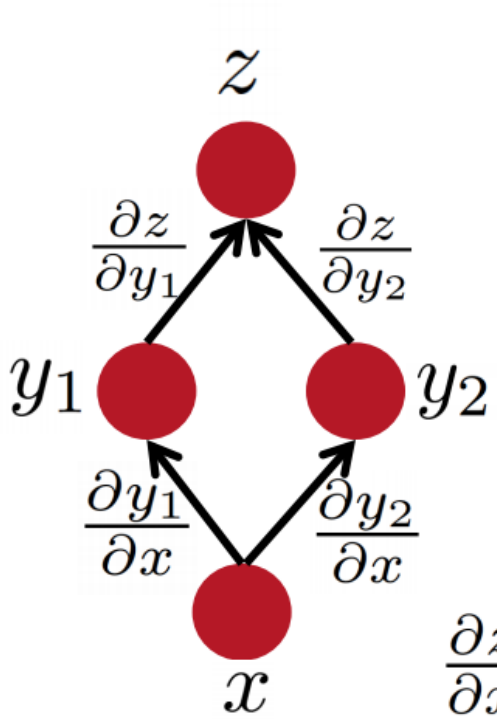
We used these facts last time when deriving the details of the Backpropagation algorithm.

Remember that *z* was the error function.

Slide Credit: Richard Socher

Some facts from real analysis

- Multiple path chain rule



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

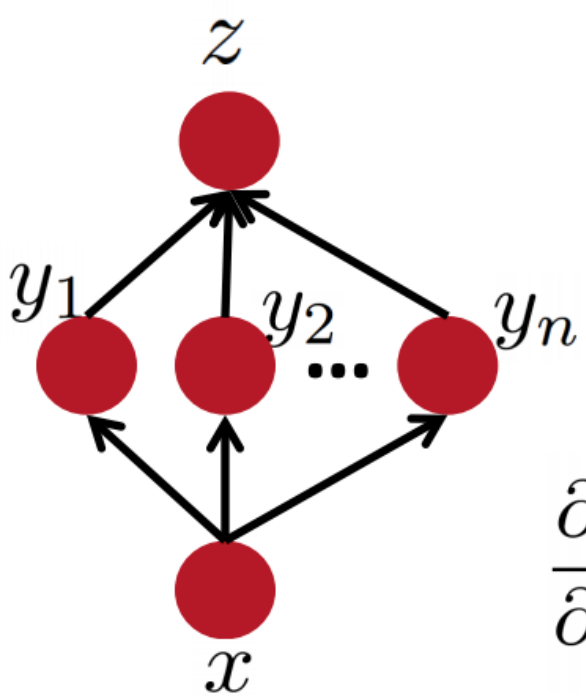
We used these facts last time when deriving the details of the Backpropagation algorithm.

Reminder

Slide Credit: Richard Socher

Some facts from real analysis

- Multiple path chain rule: general



We used these facts last time when deriving the details of the Backpropagation algorithm.

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

Reminder

Slide Credit: Richard Socher

Recap: The Backprop 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 j, 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

Gradient Checks are useful!

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

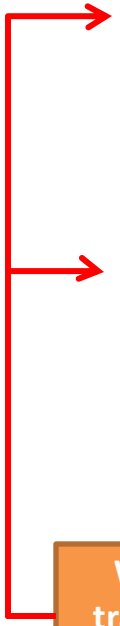
  - Implement your gradient
  - Implement a finite difference computation by looping through the parameters of your network, adding and subtracting a small epsilon (~10^-4) and estimate derivatives

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

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

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.

Sparse Auto-encoder

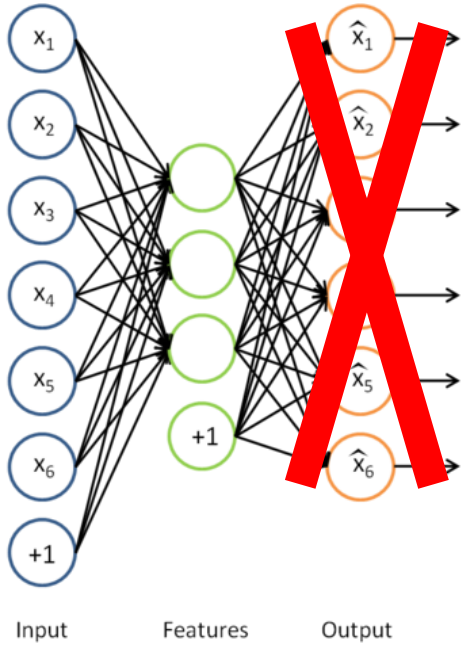
- Encoding: $\boldsymbol{y} = f(W\boldsymbol{x} + \boldsymbol{b})$
- Decoding: $\hat{\boldsymbol{x}} = g(W'\boldsymbol{y} + \boldsymbol{b}')$
  - Goal: perfect reconstruction of input vector $\boldsymbol{x}$, by the output $\hat{\boldsymbol{x}} = h_{\boldsymbol{\theta}}(\boldsymbol{x})$
    - Where $\boldsymbol{\theta}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

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

  - And regularize it

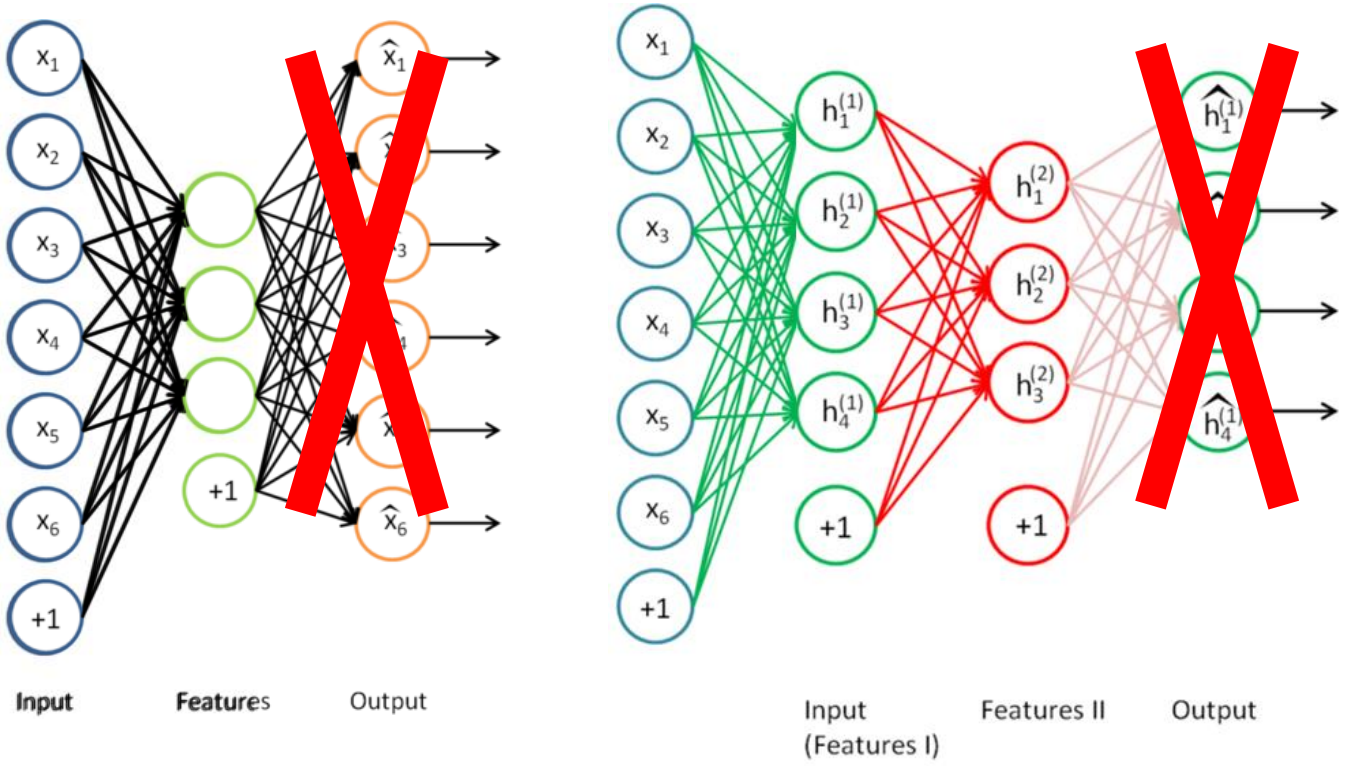
$$\min_\theta \sum_{\boldsymbol{x}} l\left(h_\theta(\boldsymbol{x}), \boldsymbol{x}\right) + \sum_i |w_i|$$

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



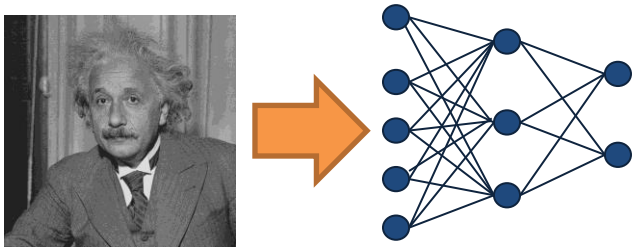
Stacking Auto-encoder

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

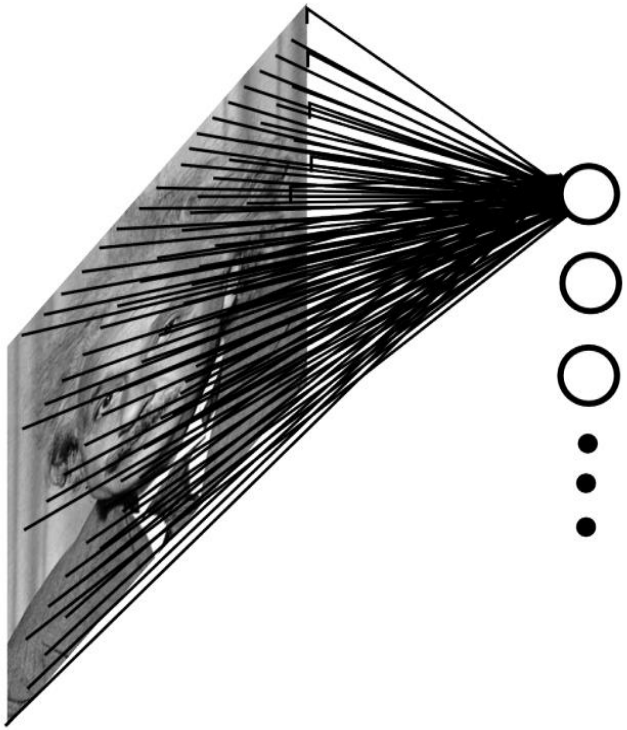


Receptive Fields

- 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?

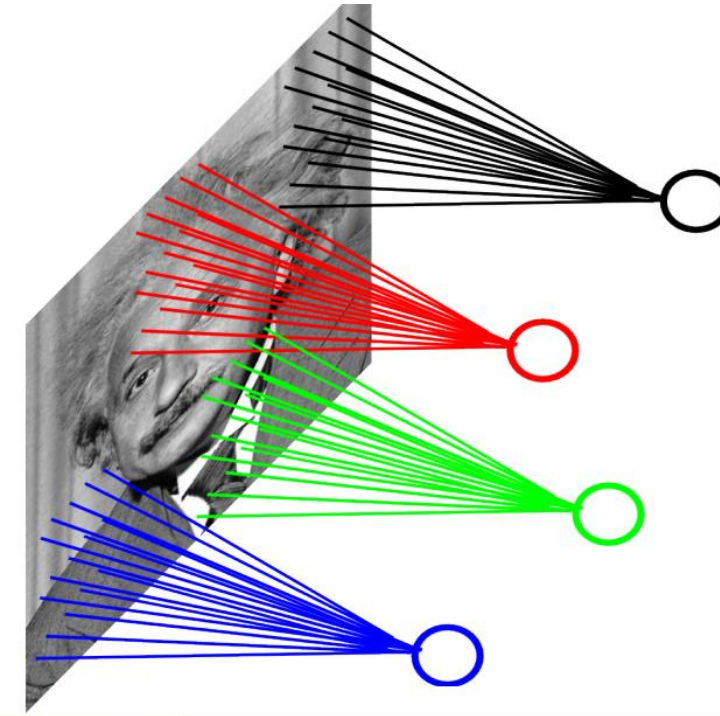


- A **fully connected layer**:
  - Example:
    - 100x100 images
    - 1000 units in the input
  - Problems:
    - 10^7 edges!
    - Spatial correlations lost!
    - Variables sized inputs.



Slide Credit: Marc'Aurelio Ranzato

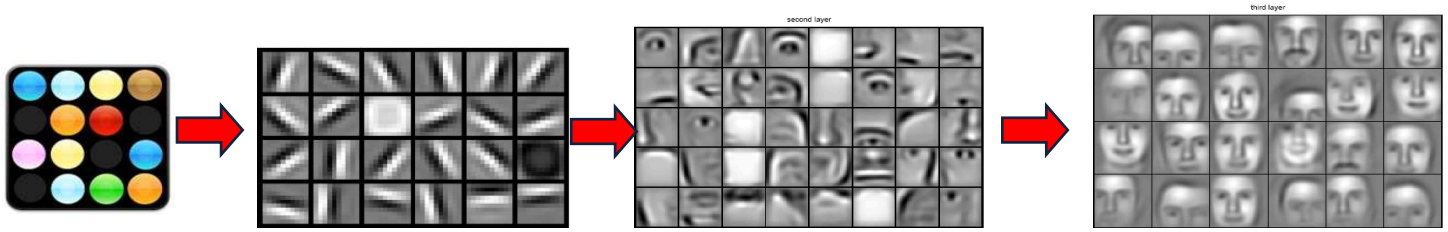
- Consider a task with image inputs:
- A **locally connected layer**:
  - Example:
    - 100x100 images
    - 1000 units in the input
    - Filter size: 10x10
  - Local correlations preserved!
  - Problems:
    - 10^5 edges
    - This parameterization is good when input image is registered (e.g., face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



So what is a convolution?

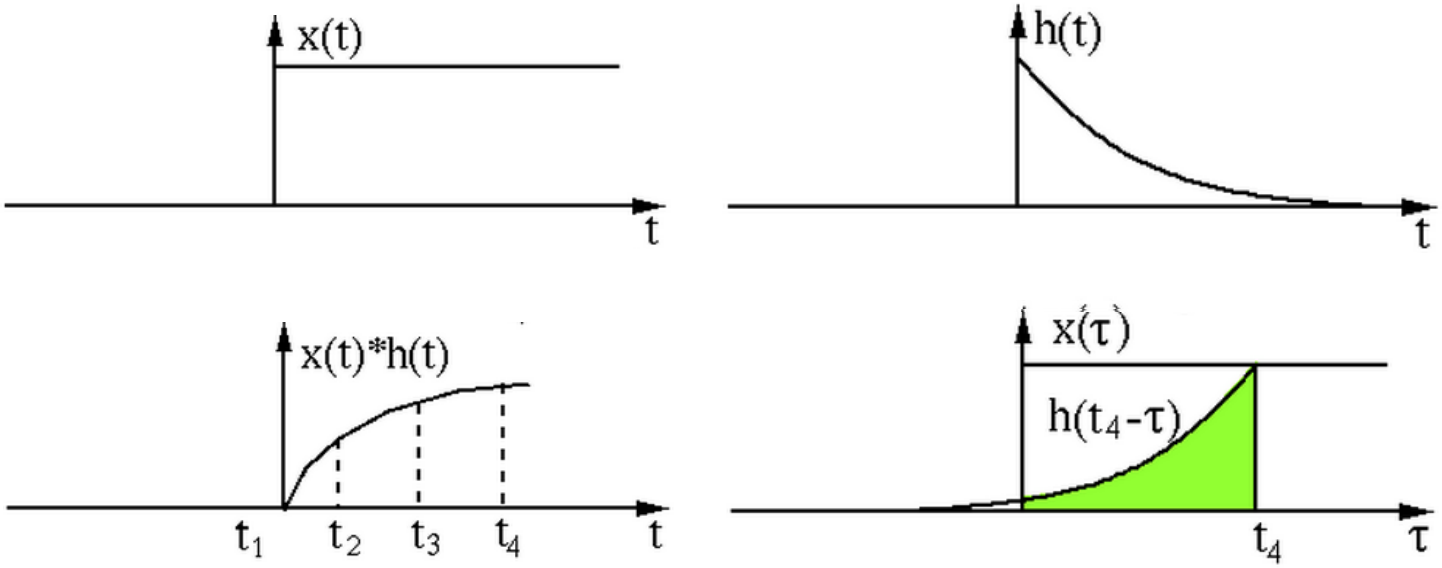
Convolution Operator

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

- Convolution operator: $*$
  - takes two functions and gives another function
- One dimension: $(x * h)(t) = \int x(\tau)h(t-\tau)d\tau$

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

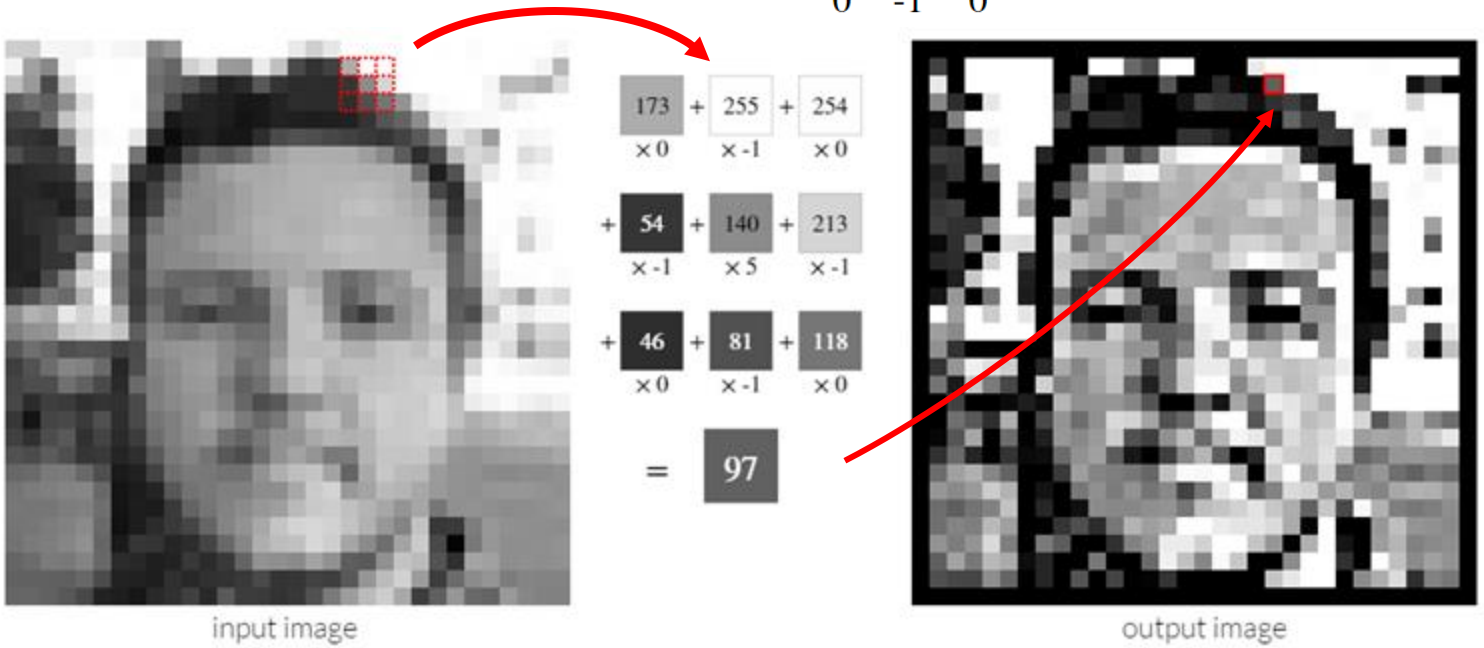
Example convolution:



Convolution Operator (2)

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

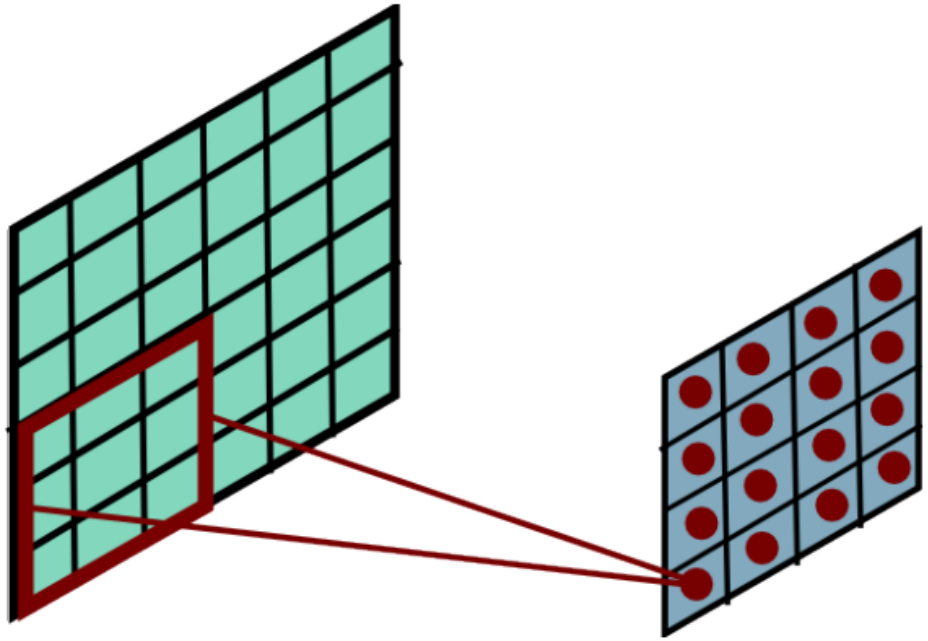
$$\begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix}$$



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

Convolution Operator (3)

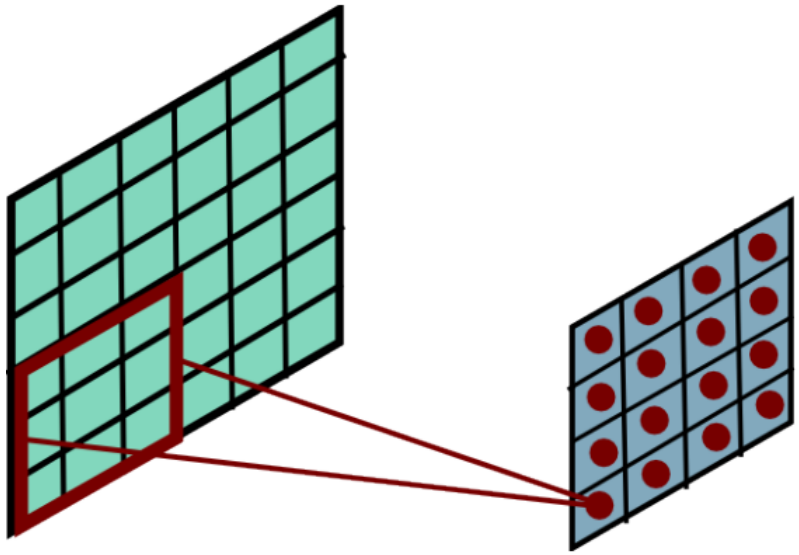
- Convolution in two dimension:
  - The same idea: flip one matrix and slide it on the other matrix



Slide Credit: Marc'Aurelio Ranzato

Complexity of Convolution

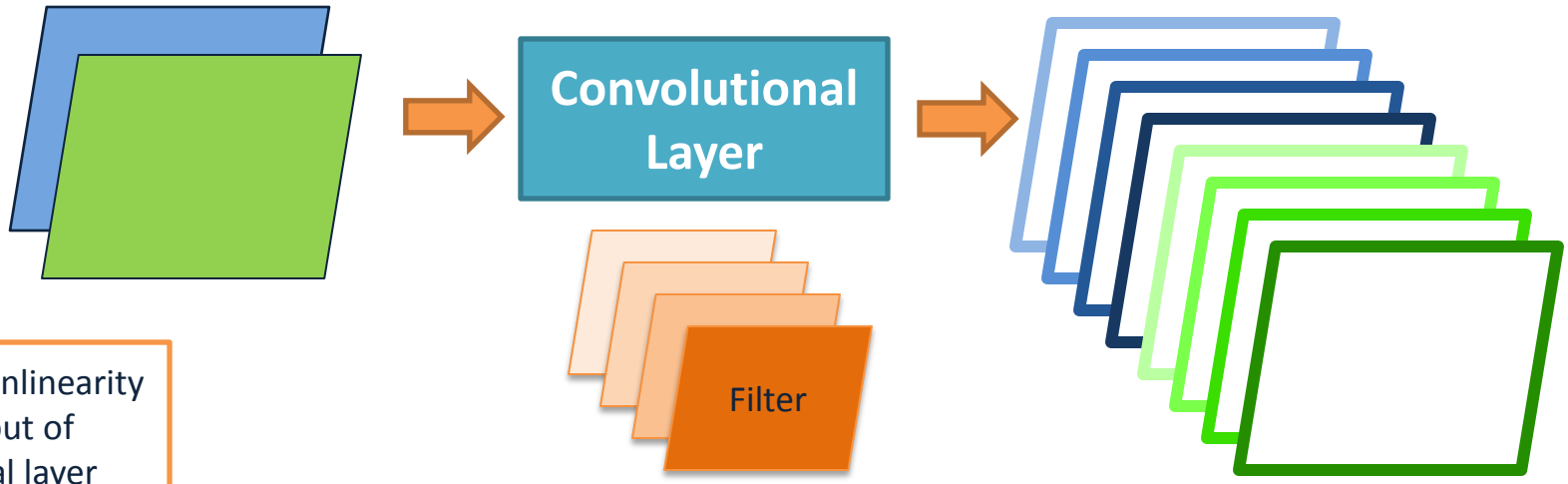
- Complexity of convolution operator is $n\log(n)$, for n inputs.
  - Uses Fast-Fourier-Transform (FFT)
- For two-dimension, each convolution takes $MN\log(MN)$ time, where the size of input is MN.



Slide Credit: Marc'Aurelio Ranzato

Convolutional Layer

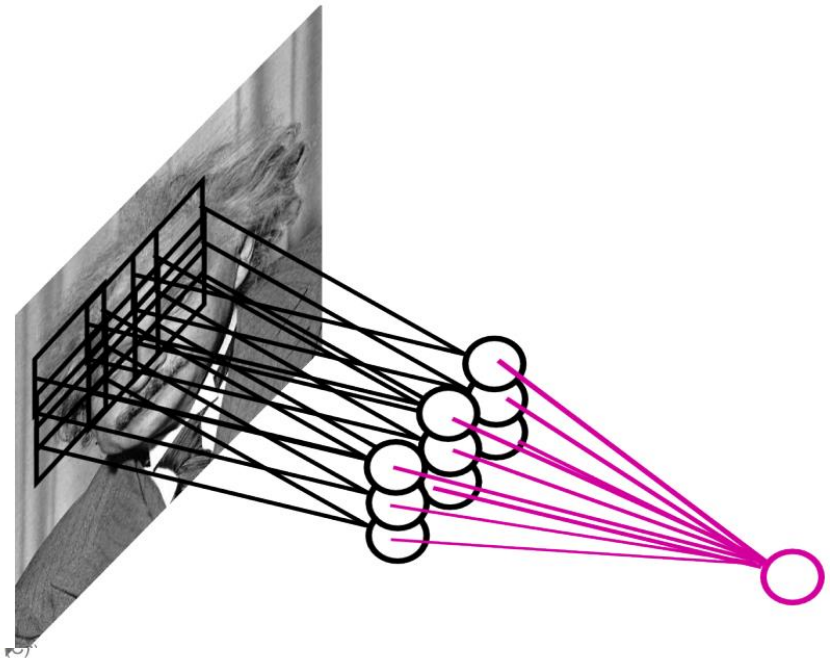
- The convolution of the **input (vector/matrix)** with weights **(vector/matrix)** results in a **response vector/matrix**.
- We can have **multiple filters** in each convolutional layer, each producing an output.
- If it is an intermediate layer, it can have **multiple inputs**!



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

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

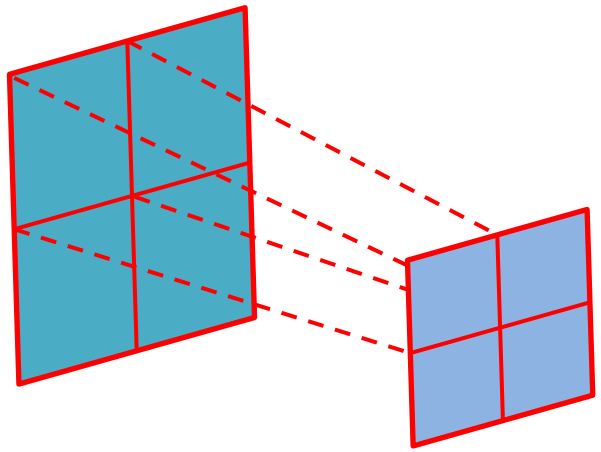
    - Average pooling

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

    - L2-pooling

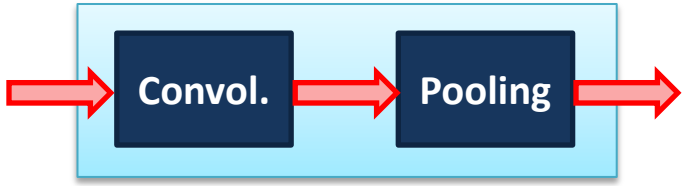
$$h_i[n] = \frac{1}{n} \sqrt{\sum_{i \in N(n)} \tilde{h}^2[i]}$$

    - etc

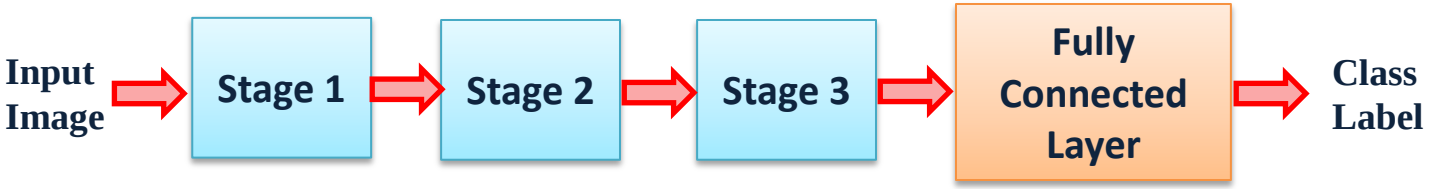


Convolutional Nets

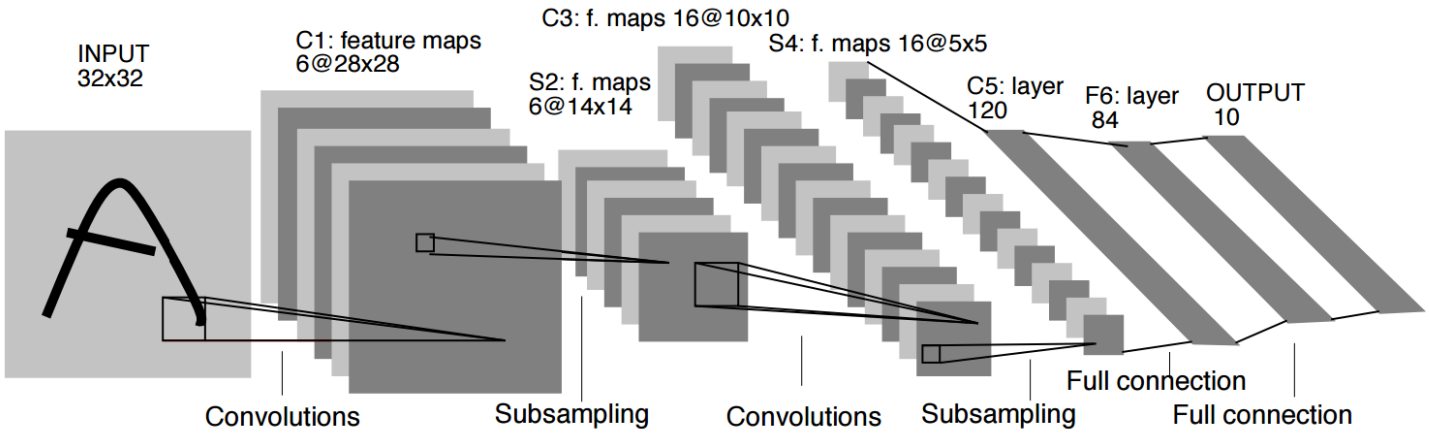
- One stage structure:



- Whole system:

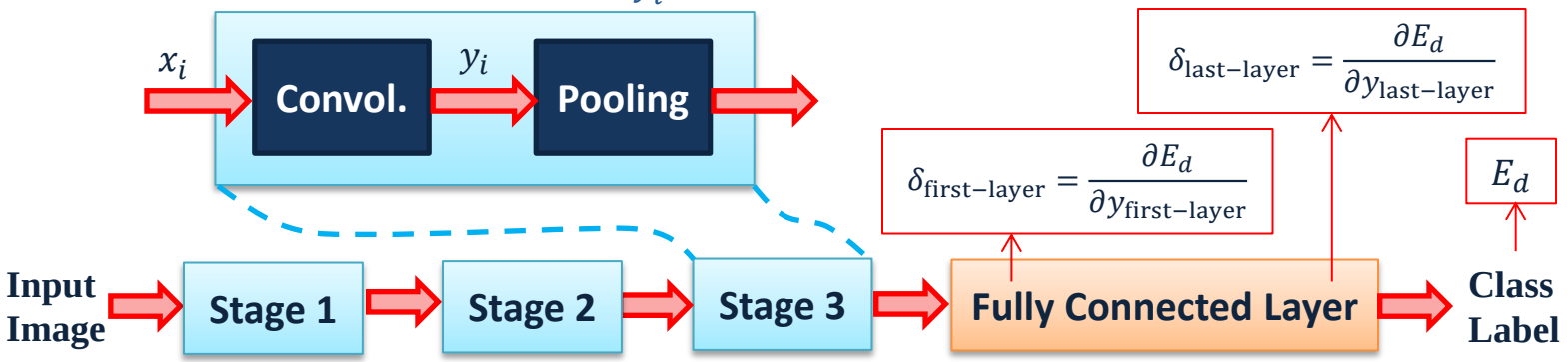


An example system :



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:

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

The convolution

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

A differentiable nonlinearity

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

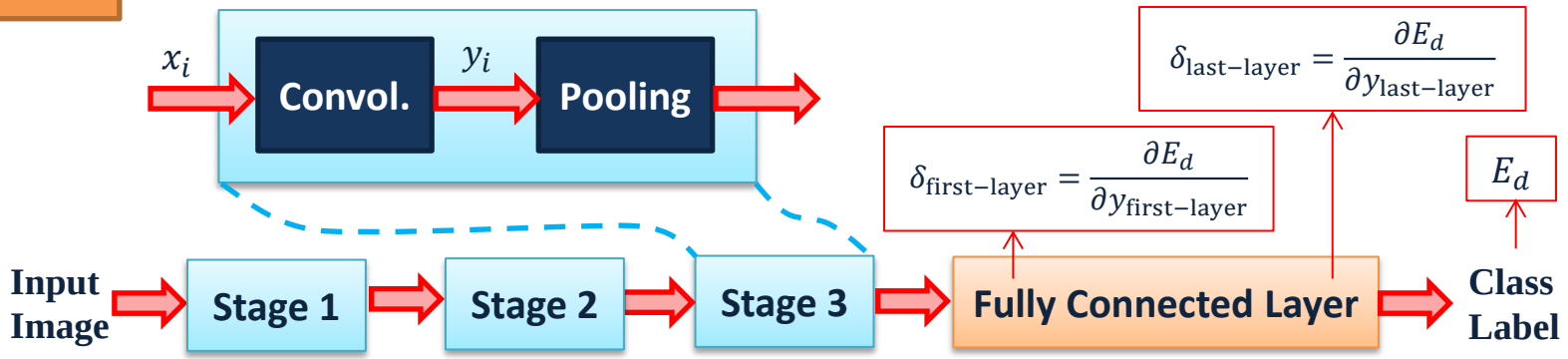
$$\frac{\partial E_d}{\partial \tilde{y}_i} = \frac{\partial E_d}{\partial y_i} \frac{\partial y_i}{\partial \tilde{y}_i} = \frac{\partial E_d}{\partial y_i} f'(\tilde{y})$$

Now we have everything in this layer to update the filter

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

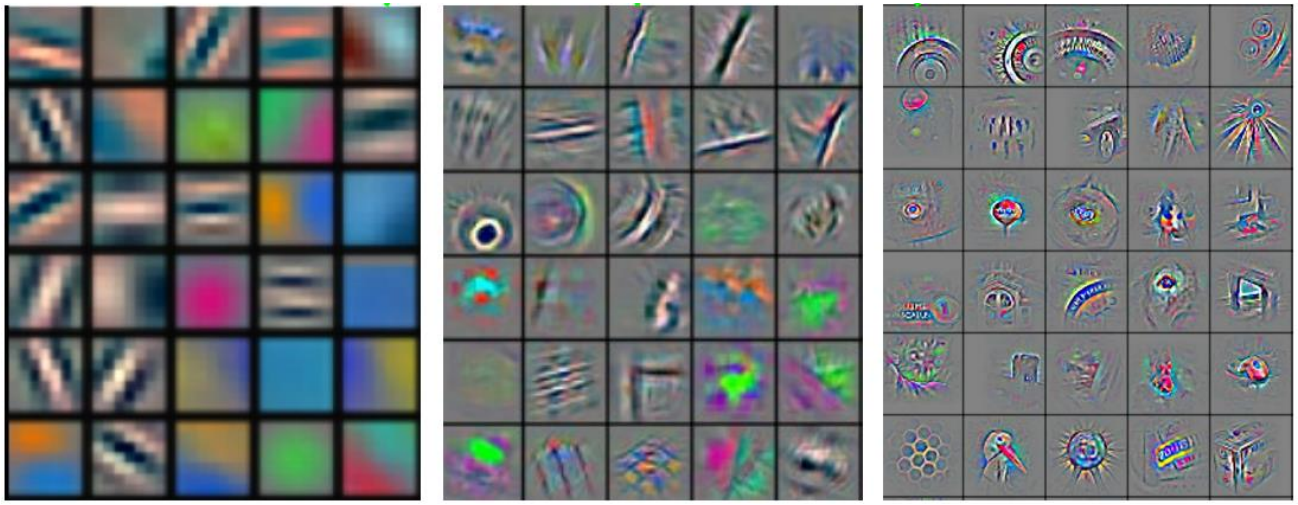
We need to pass the gradient to the previous layer

Now we can repeat this for each stage of ConvNet.



Convolutional Nets

Stage 1 → Stage 2 → Stage 3 → Fully Connected Layer → Class Label

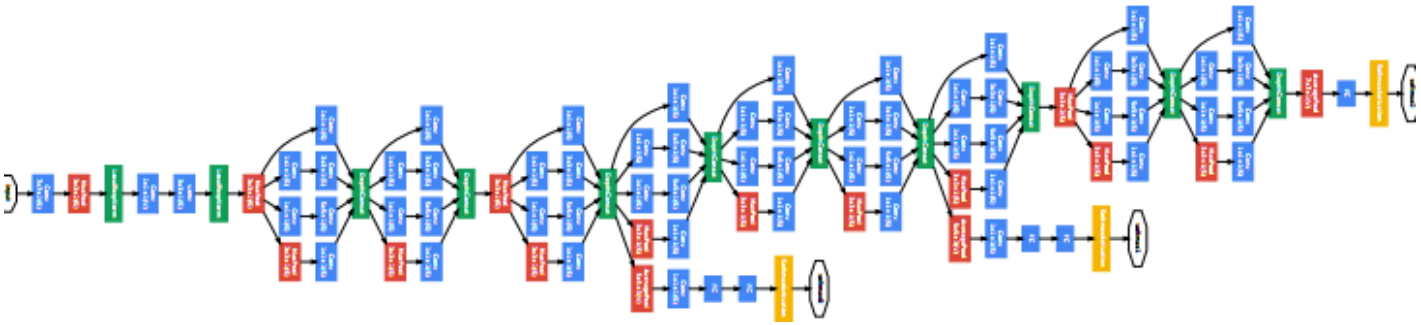


An example system :

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
    - Compute different filters
    - Compose one big vector from all of them
    - Layer this iteratively



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

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.

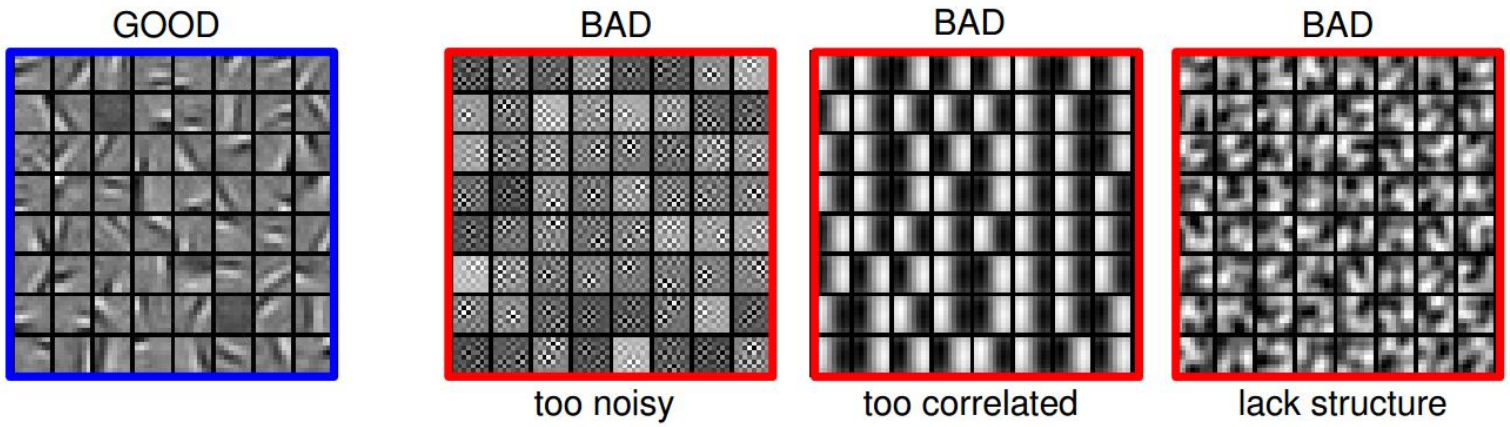


Figure Credit: Marc'Aurelio Ranzato

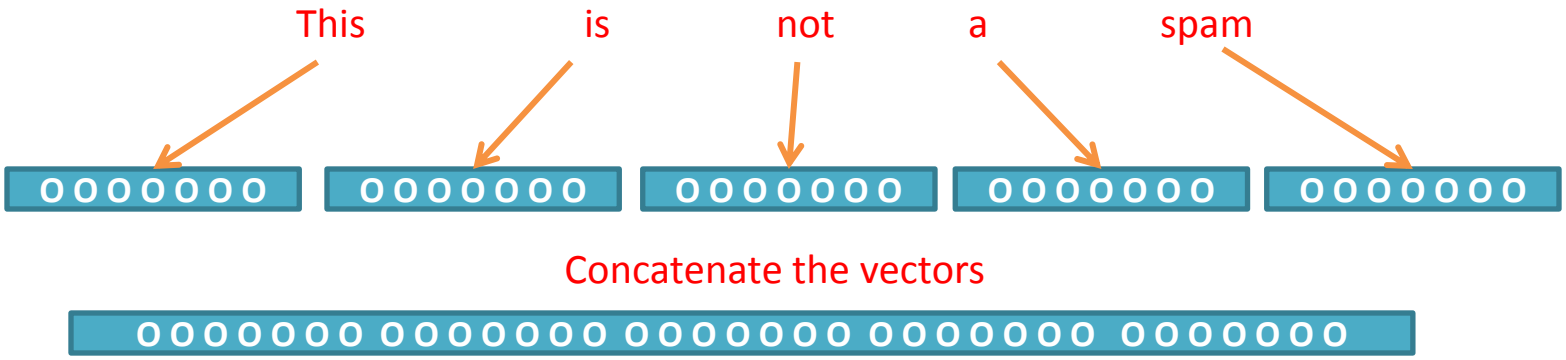
Debugging

- 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 $\mathbb{R}^d$

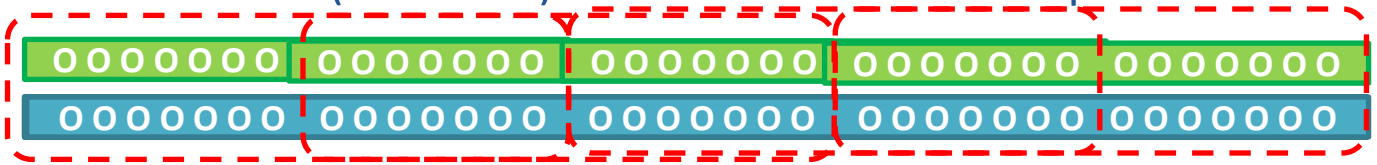


- Now we can assume that the input to the system is a vector $\mathbb{R}^{dl}$
  - Where the input sentence has length l ($l = 5$ in our example)
  - Each word vector's length d ($d = 7$ in our example)

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



$$c_1 = f(w.x_{1:2h}) \quad f(w.x_{h+1:3h}) \quad f(w.x_{2h+1:4h}) \quad f(w.x_{3h+1:4h})$$

  - 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)

A convolutional layer

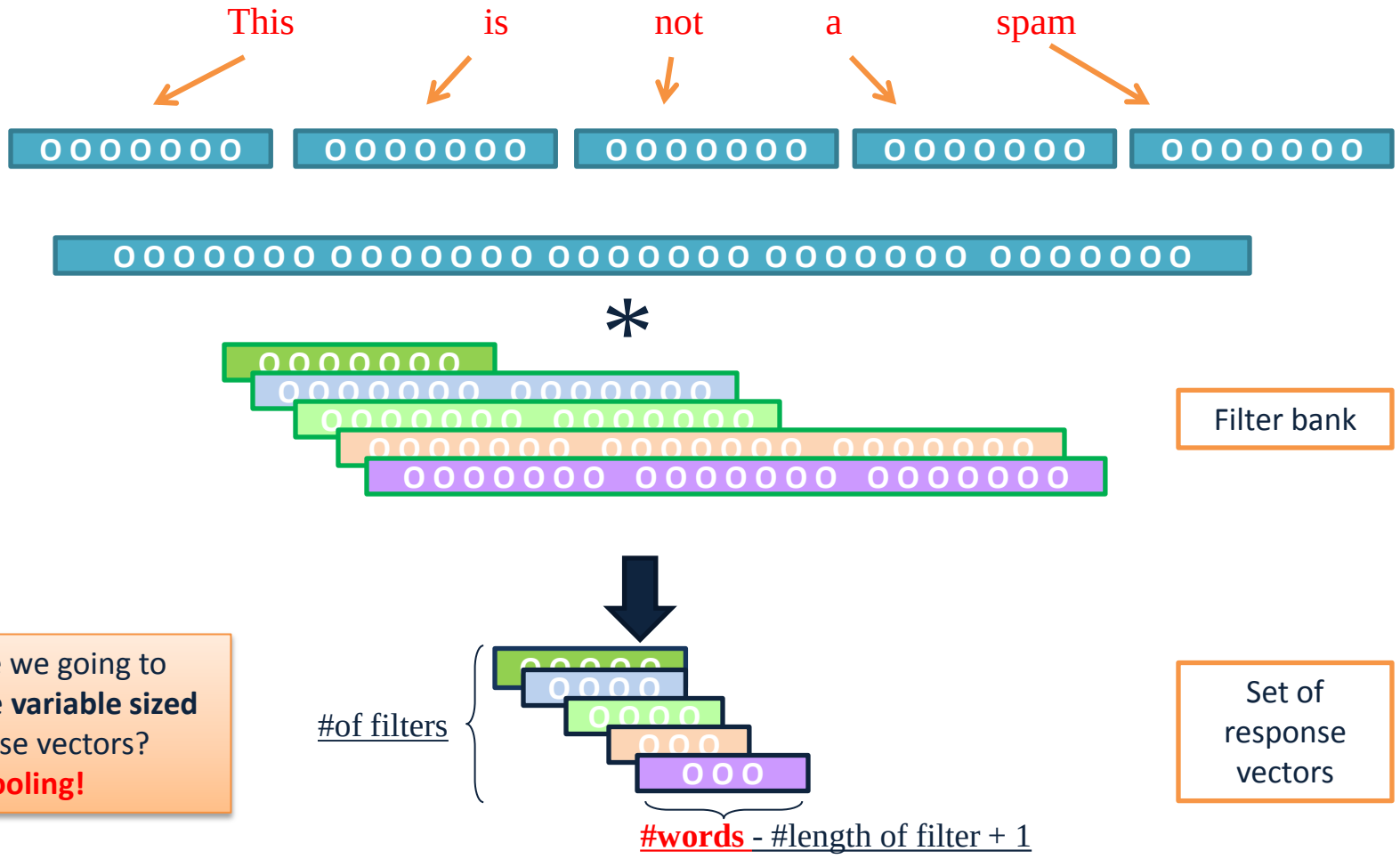
Convolutional Layer on vectors

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!**



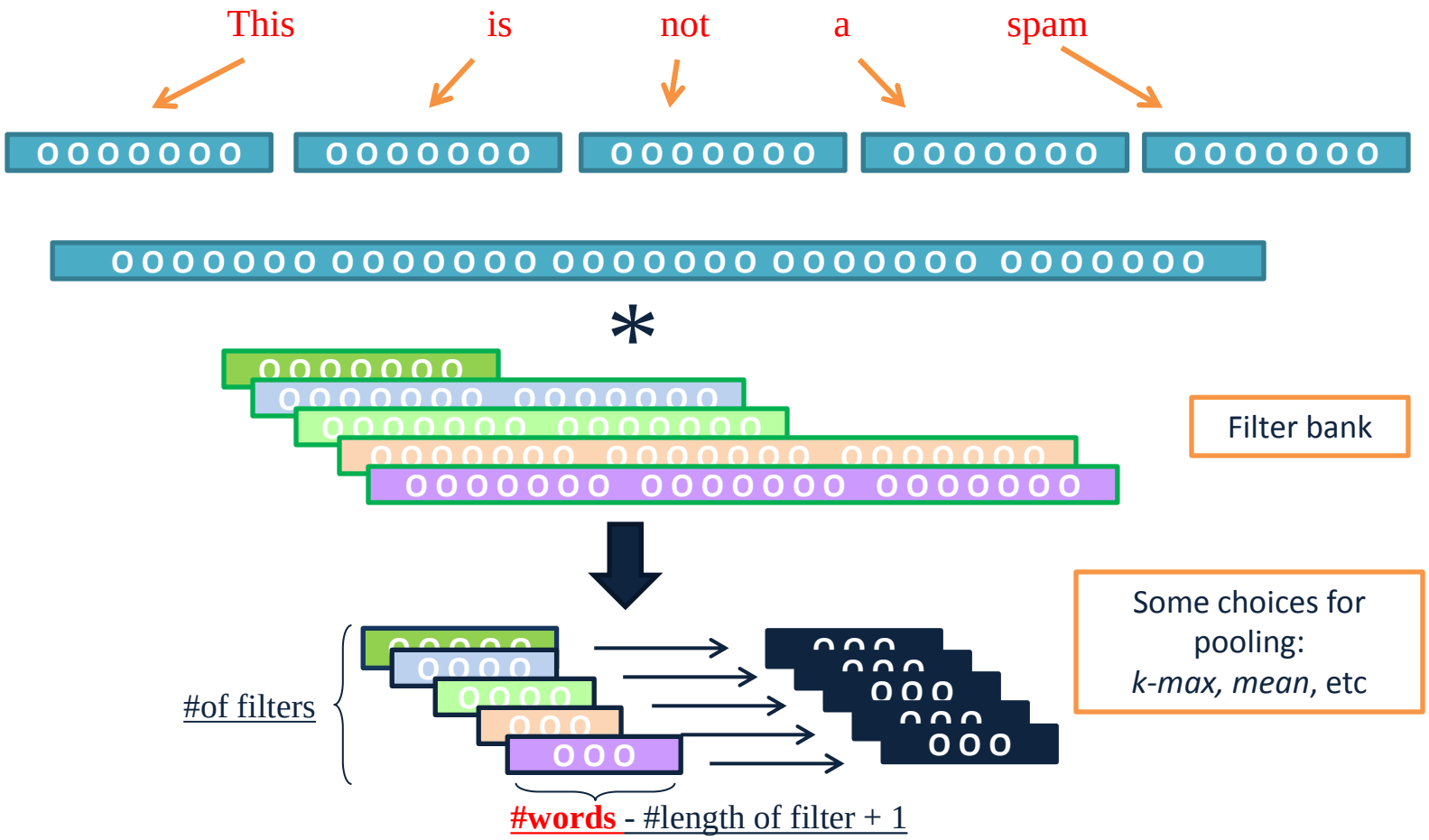
Convolutional Layer on vectors

Get word vectors for each words

Concatenate vectors

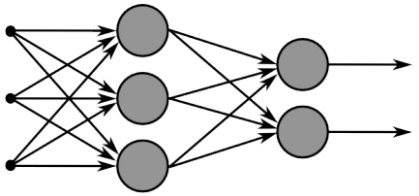
Perform convolution with each filter

Pooling on filter responses

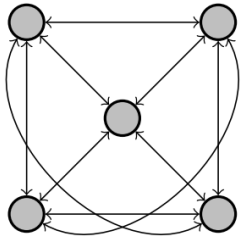


- Now we can pass the fixed-sized vector to a logistic unit (softmax), or give it to multi-layer network (last session)

Recurrent Neural Networks



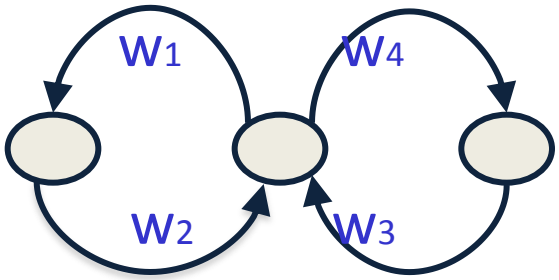
- Multi-layer feed-forward NN: **DAG**
  - Just computes a fixed sequence of non-linear learned transformations to convert an input patter into an output pattern

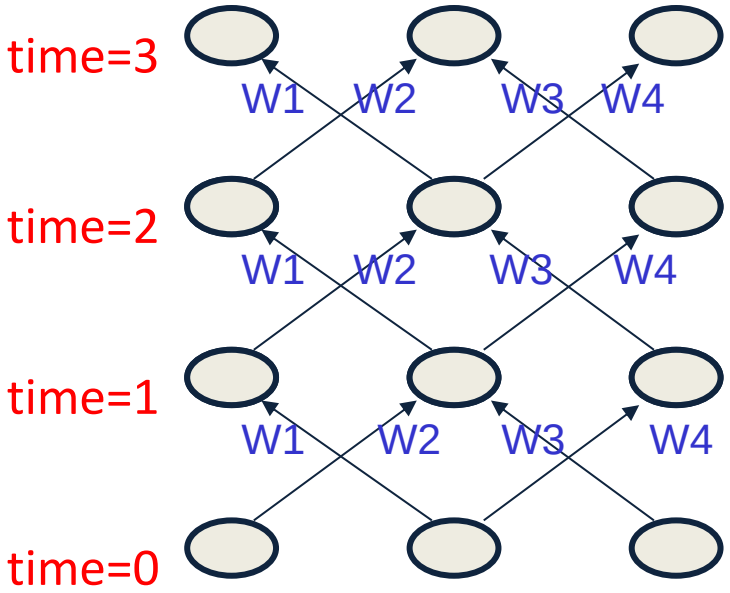


- Recurrent Neural Network: **Digraph**
  - Has cycles.
  - Cycle can act as a memory;
  - The hidden state of a recurrent net can carry along information about a "potentially" unbounded number of previous inputs.
  - They can model sequential data in a much more natural way.

Equivalent between RNN and Feed-forward NN

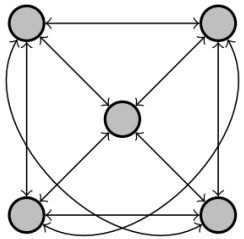
- Assume that there is a time delay of 1 in using each connection.
- The recurrent net is just a layered net that keeps reusing the same weights.





Slide Credit: Geoff Hinton

Recurrent Neural Networks



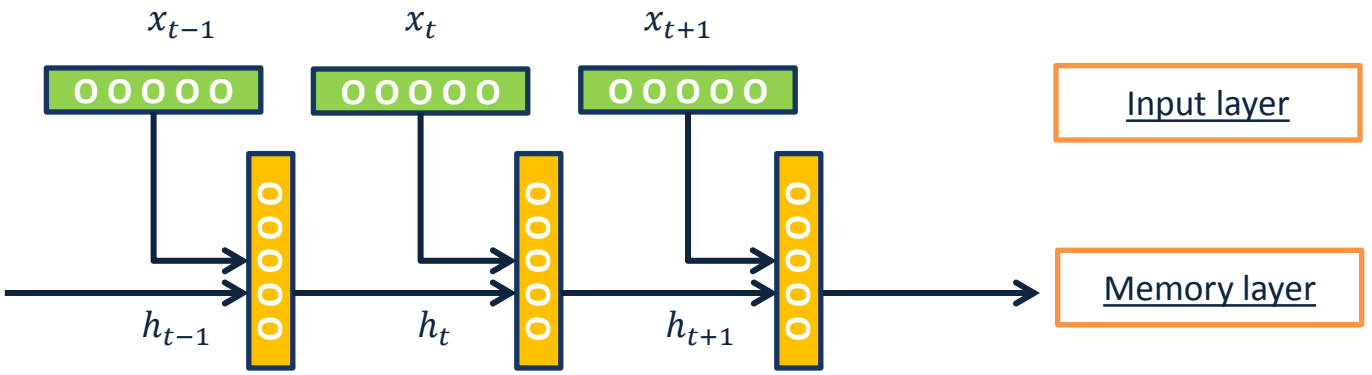
- Training a general RNN's can be hard
  - Here we will focus on a **special family of RNN's**
- Prediction on chain-like input:
  - Example: POS tagging words of a sentence

$X =$	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.

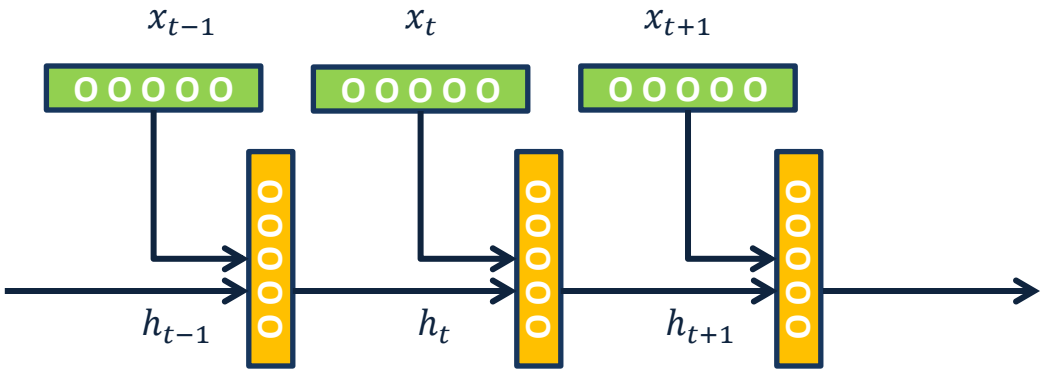


Recurrent Neural Networks

- 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

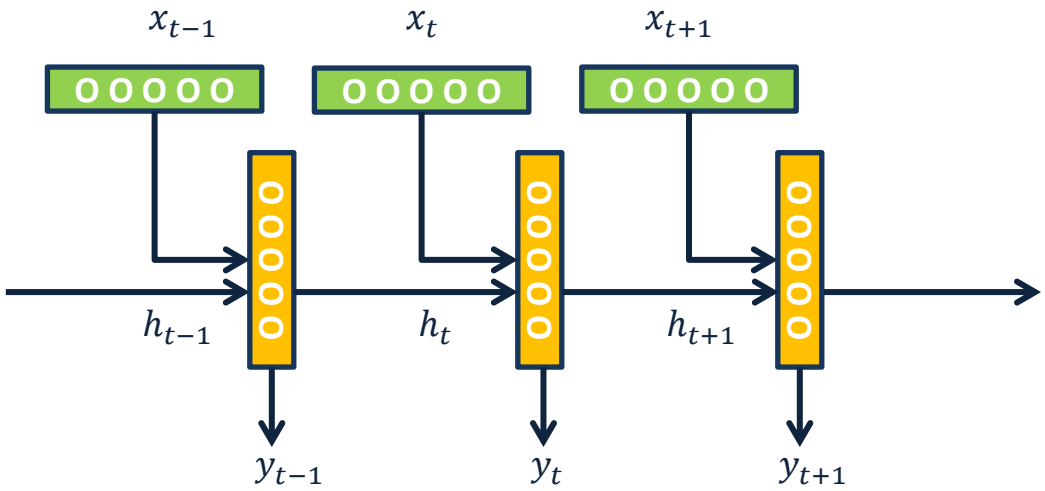
$$y_t = \text{softmax}(W_o h_t)$$

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

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

- Prediction for the whole chain

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



- 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

Training RNNs

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

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

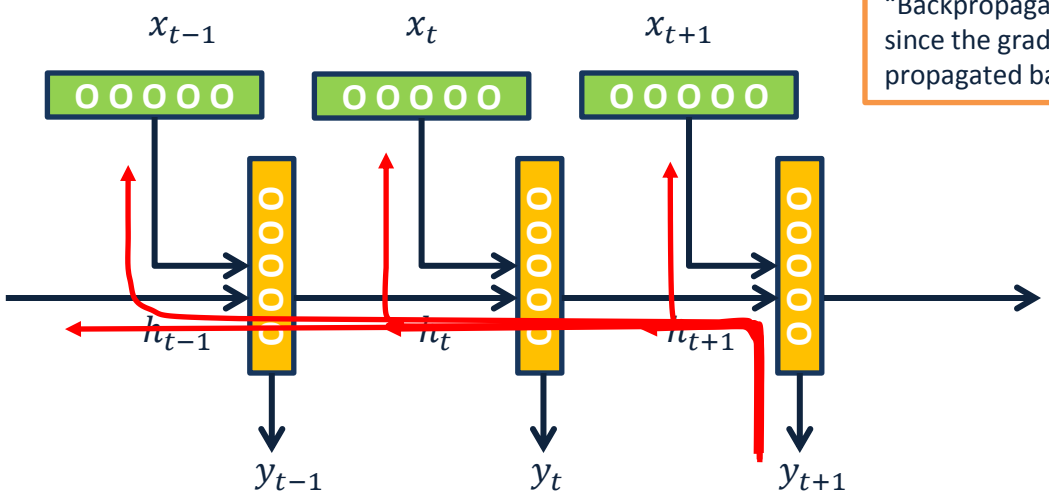
$$\frac{\partial E_t}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t-k}} \frac{\partial h_{t-k}}{\partial W}$$

Parameters? W_o, W_i, W_h + vectors for input

Backpropagation for RNN

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

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

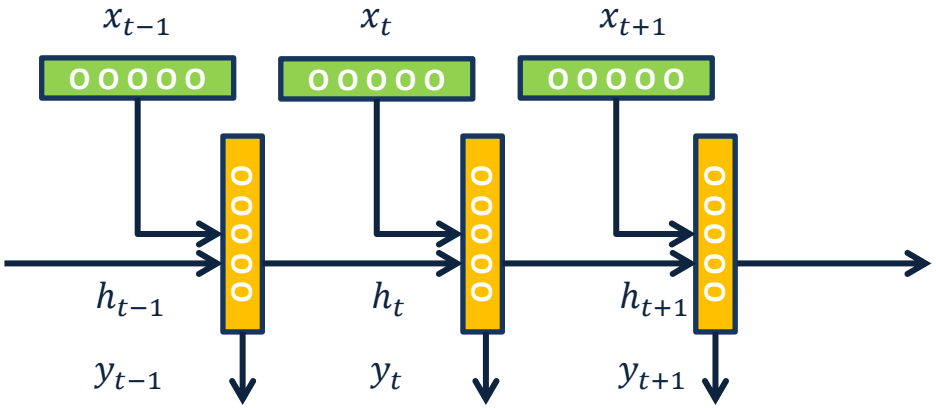
Reminder:
$y_t = \text{softmax}(W_o h_t)$
$h_t = f(W_h h_{t-1} + W_i x_t)$

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

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

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

Backpropagation for RNN



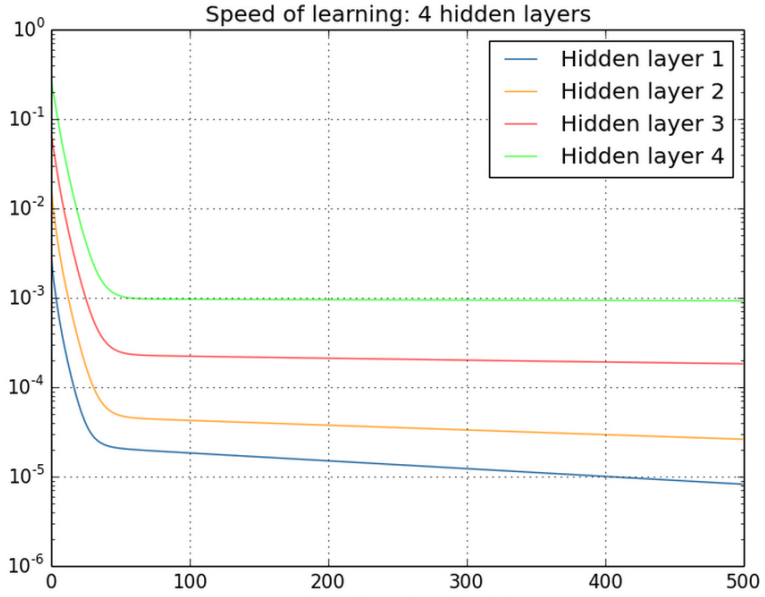
Vanishing/exploding gradients

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

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

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.

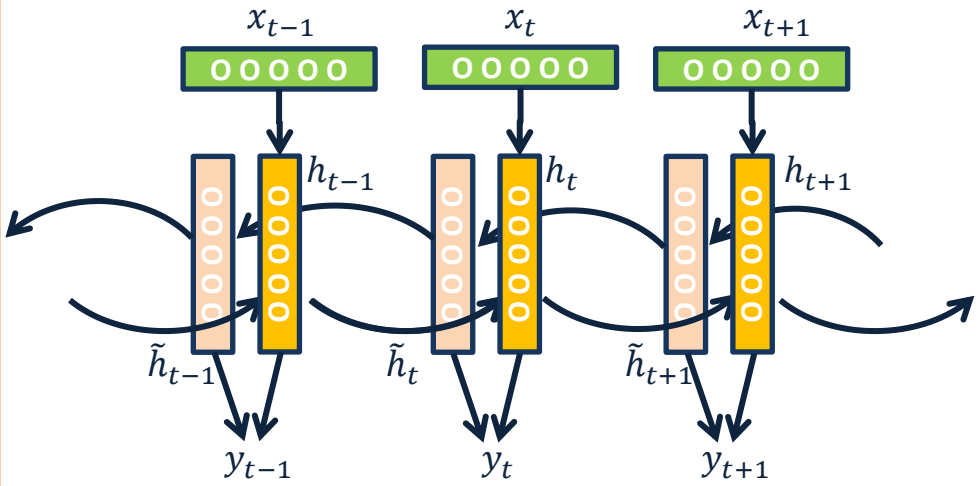


Vanishing/exploding gradients

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

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



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

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

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

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