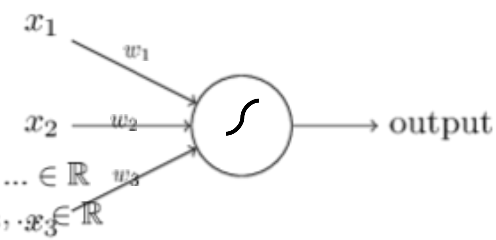
CS60020: Foundations of Algorithm Design and Machine Learning

Sourangshu Bhattacharya

Neural Network Basics

- Given several **inputs**:
  and several **weights**:
  and a **bias** value:



$x_1, x_2, x_3, \ldots \in \mathbb{R}$
$w_1, w_2, w_3, \ldots \in \mathbb{R}$
$b \in \mathbb{R}$

- A neuron produces a single output:

$$o_1 = s(\textstyle\sum_i w_i x_i + b)$$

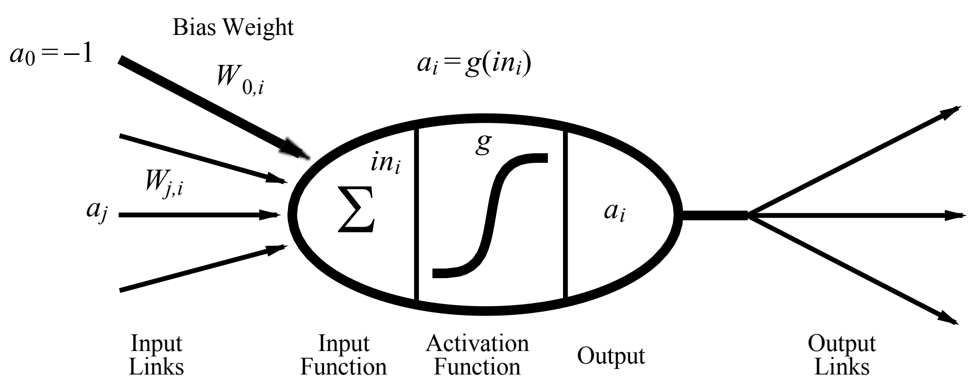
$$\textstyle\sum_i w_i x_i + b$$

- This sum is called the **activation** of the neuron
- The function *s* is called the **activation function** for the neuron
- The weights and bias values are typically initialized randomly and learned during training

McCulloch–Pitts “unit”

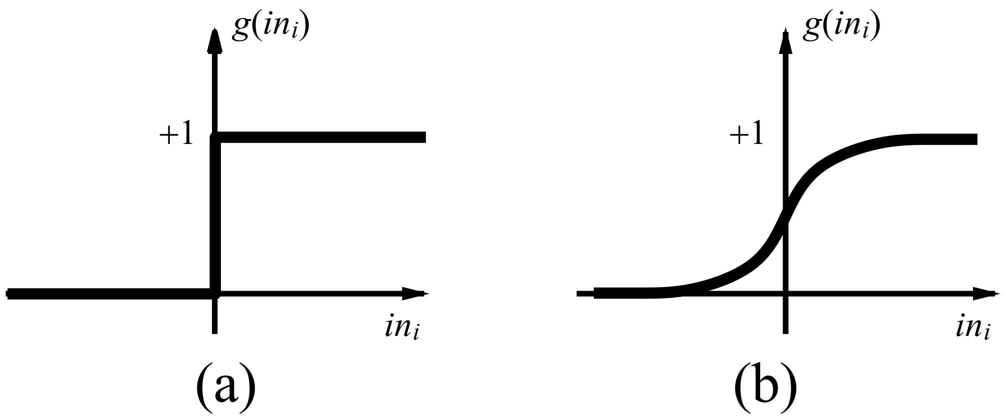
Output is a “squashed” linear function of the inputs:

$$a_i \leftarrow g(in_i) = g\left(\Sigma_j W_{j,i} a_j\right)$$



A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

Activation functions

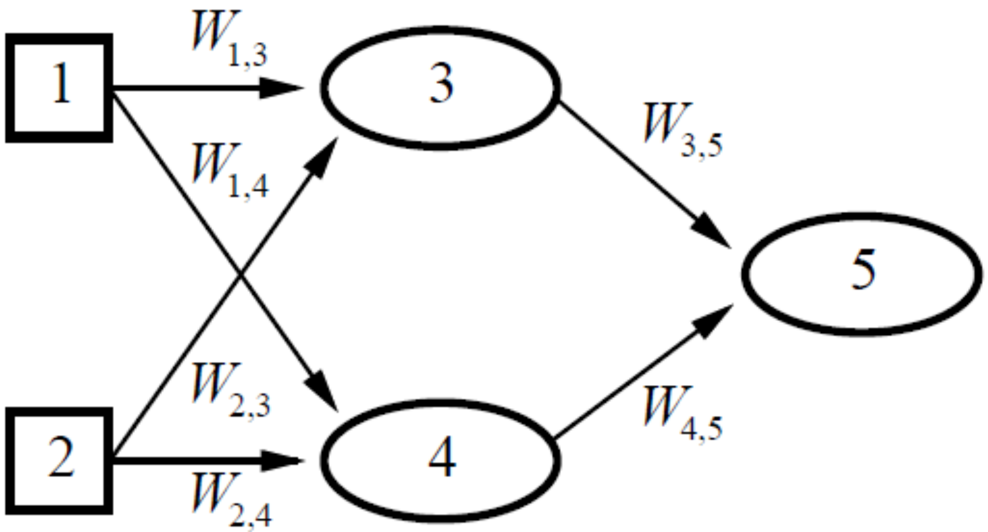


(a)is a step function or threshold function

(b)is a sigmoid function $1/(1+e^{-x})$

Changing the bias weight $W_{0,i}$ moves the threshold location

Feed forward example



Feed-forward network = a parameterized family of nonlinear functions:

$$
\begin{aligned}
a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\
&= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2))
\end{aligned}
$$

Adjusting weights changes the function: do learning this way!

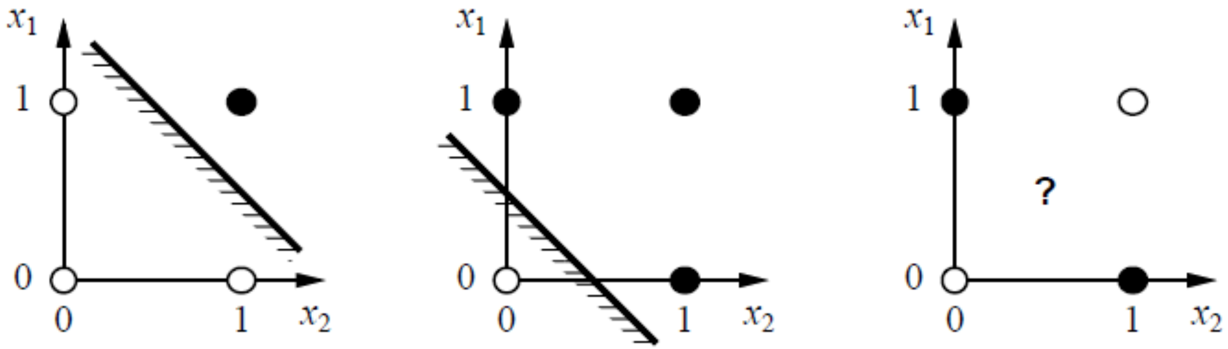
Expressiveness of perceptrons

Consider a perceptron with g = step function (Rosenblatt, 1957, 1960)

Can represent AND, OR, NOT, majority, etc., but not XOR

Represents a linear separator in input space:

$\Sigma_j W_j x_j > 0$ or $\mathbf{W} \cdot \mathbf{x} > 0$

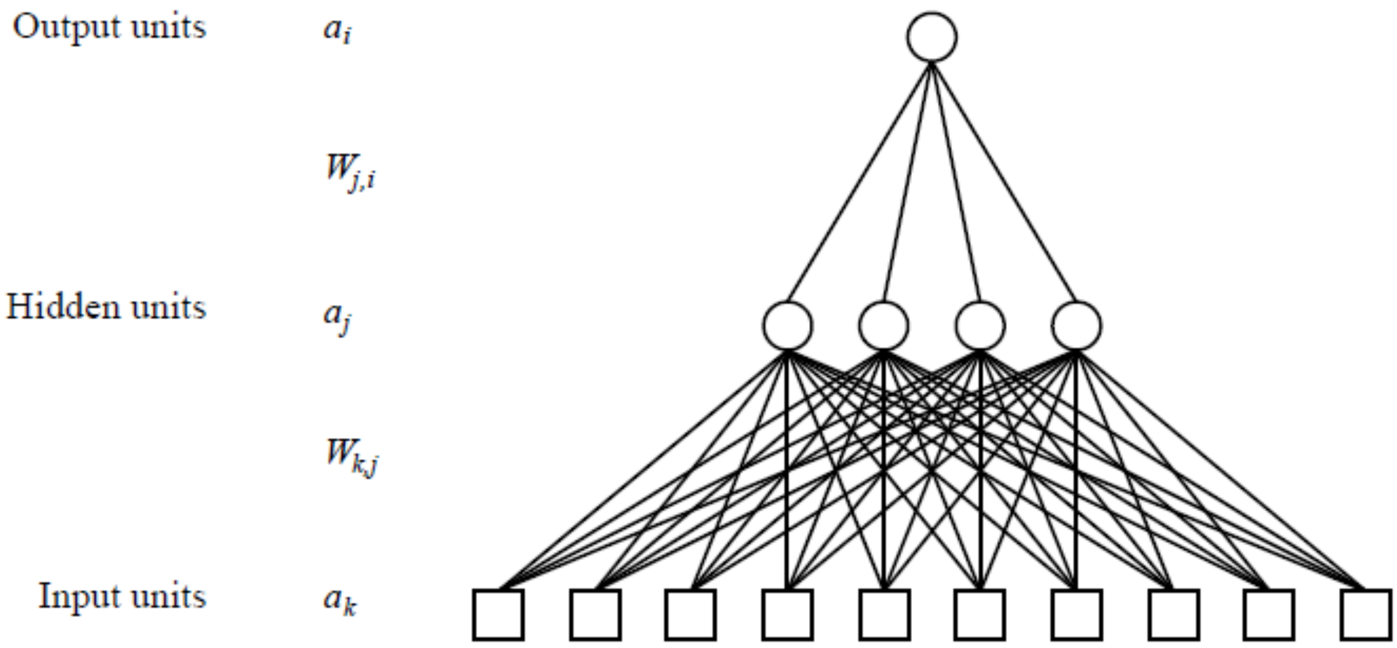


(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

Minsky & Papert (1969) pricked the neural network balloon

Feed Forward Neural Networks

Layers are usually fully connected;
numbers of hidden units typically chosen by hand



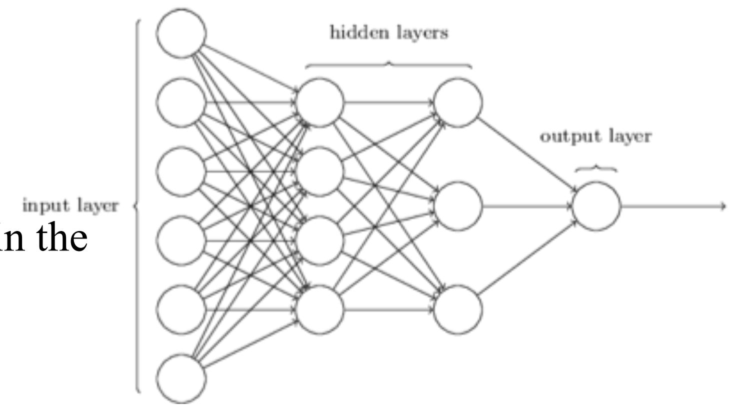
Hidden-Layer

- The hidden layer (L_2, L_3) represent learned non-linear combination of input data
- For solving the XOR problem, we need a hidden layer
  - some neurons in the hidden layer will activate only for some combination of input features
  - the output layer can represent combination of the activations of the hidden neurons
- Neural network with one hidden layer **is a universal approximator**
  - Every function can be modeled as a shallow feed forward network
  - Not all functions can be represented *efficiently* with a single hidden layer $\Rightarrow$ we still need deep neural networks

Going from Shallow to Deep Neural Networks

- Neural Networks can have several hidden layers
- Initializing the weights randomly and training all layers at once does hardly work
- Instead we train layerwise on unannotated data (a.k.a. pre-training):
  - Train the first hidden layer
  - Fix the parameters for the first layer and train the second layer.
  - Fix the parameters for the first & second layer, train the third layer

Img-Source: http://neuralnetworksanddeeplearning.com

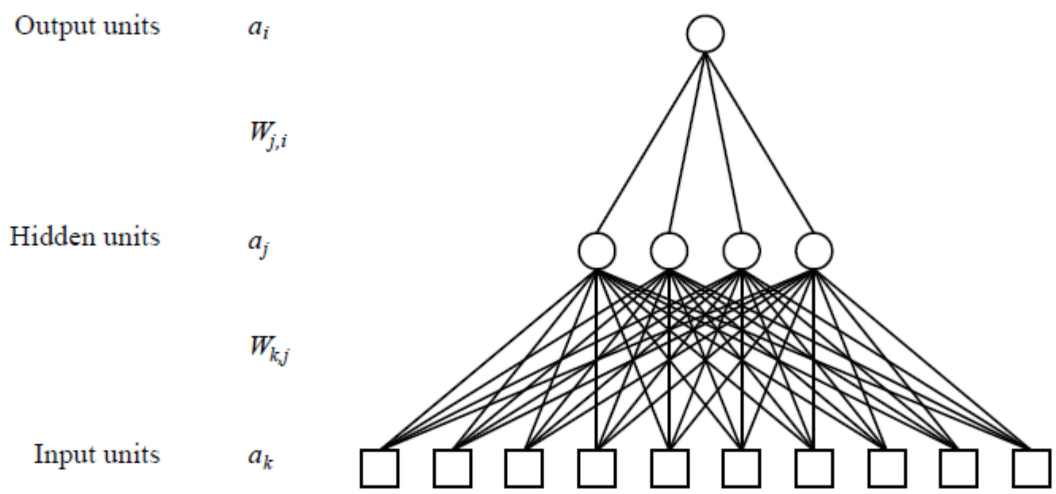


- After the pre-training, train all layers using your annotated data
- The pre-training on your unannotated data creates a high-level abstractions of the input data
- The final training with annotated data fine tunes all parameters in the network

How to learn the weights

- Initialise the weights i.e. $W_{k,j}$ $W_{j,i}$ with random values
- With input entries we calculate the predicted output
- We compare the prediction with the true output
- The error is calculated
- The error needs to be sent as feedback for updating the weights

Layers are usually fully connected;
numbers of hidden units typically chosen by hand



Window Classification

Example: Classify 'Paris' in the context of this sentence with window length 2:

… museums in Paris are amazing … .

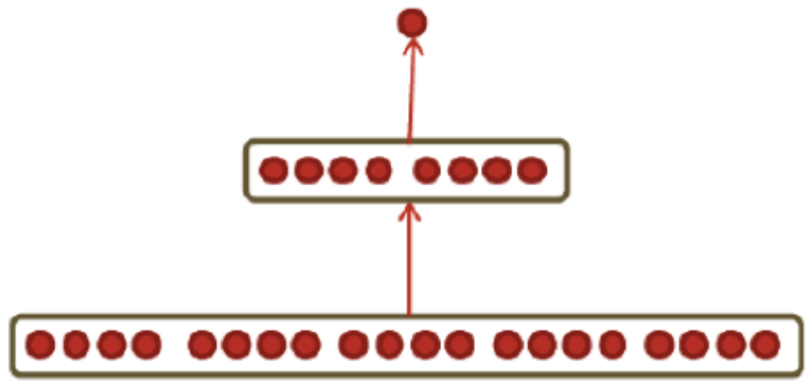
$$X_{window} = [\ x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}\ ]^T$$

Resulting vector $x_{window} \in R^{5d}$ is a column vector.

Feed-forward computation

$$s = U^T f(Wx + b) \qquad x \in \mathbb{R}^{20\times 1}, W \in \mathbb{R}^{8\times 20}, U \in \mathbb{R}^{8\times 1}$$

$$\begin{aligned} s &= U^T a \\ a &= f(z) \\ z &= Wx + b \end{aligned}$$



x_{window} = [$x_{museums}$ x_{in} x_{Paris} x_{are} $x_{amazing}$]

Maximum Margin Objective Function

Idea

Ensure that the score computed for “true” labeled data points is higher than the score computed for “false” labeled data points.

- s = score(museums in Paris are amazing)
- s_c = score(Not all museums in Paris)

Maximum Margin Objective Function

Objective

Maximize $(s - s_c)$ or to minimize $(s_c - s)$. One possible objective function:
minimize $J = max(s_c - s, 0)$

What is the problem with this?

- Does not attempt to create a margin of safety. We would want the "true" labeled data point to score higher than the "false" labeled data point by some positive margin Δ.
- We would want error to be calculated if $(s - s_c < \Delta)$ and not just when $(s - s_c < 0)$. The modified objective:
  minimize $J = max(\Delta + s_c - s, 0)$

Maximum Margin Objective Function

- Objective for a single window: $J = max(1 + s_c - s, 0)$
- Each window with a location at its center should have a score +1 higher than any window without a named entity at its center.
- $s = U^T f(Wx+b)$, $s_c = U^T f(Wx_c+b)$
- Assuming cost J is > 0, compute the derivatives of s and s_c with respect to the involved variables: U, W, b, x

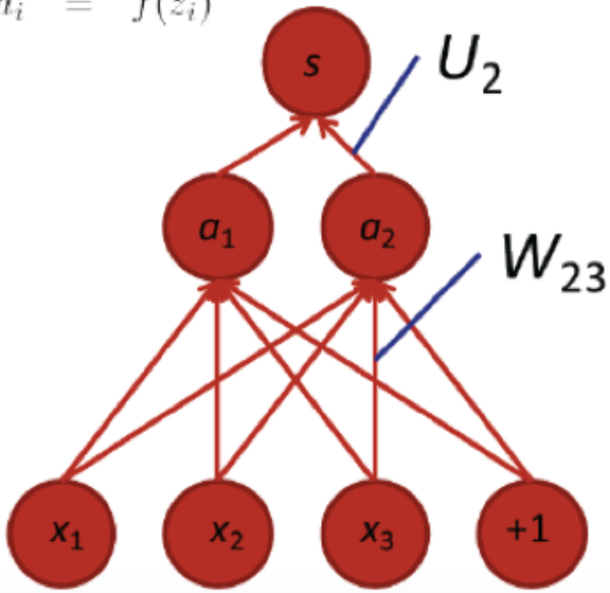
Training with backpropagation

Derivative of weight W_{ij}:

$$\frac{\partial}{\partial W_{ij}} U^T a \quad \rightarrow \quad \frac{\partial}{\partial W_{ij}} U_i a_i$$

$$\begin{aligned}
U_i \frac{\partial}{\partial W_{ij}} a_i &= U_i \frac{\partial a_i}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}} \\
&= U_i \frac{\partial f(z_i)}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}} \\
&= U_i f'(z_i) \frac{\partial z_i}{\partial W_{ij}} \\
&= U_i f'(z_i) \frac{\partial W_{i\cdot} x + b_i}{\partial W_{ij}}
\end{aligned}$$

$$\begin{aligned}
z_i &= W_{i\cdot} x + b_i = \sum_{j=1}^{3} W_{ij} x_j + b_i \\
a_i &= f(z_i)
\end{aligned}$$



Derivative continued ...

$$
\begin{aligned}
U_i \frac{\partial}{\partial W_{ij}} a_i &= U_i f'(z_i) \frac{\partial W_{i\cdot} x + b_i}{\partial W_{ij}} \\
&= U_i f'(z_i) \frac{\partial}{\partial W_{ij}} \sum_k W_{ik} x_k \\
&= \underbrace{U_i f'(z_i)}\, x_j \\
&= \underbrace{\delta_i}_{\text{Local error signal}} \quad \underbrace{x_j}_{\text{Local input signal}}
\end{aligned}
$$

where $f'(z) = f(z)(1 - f(z))$ for logistic f

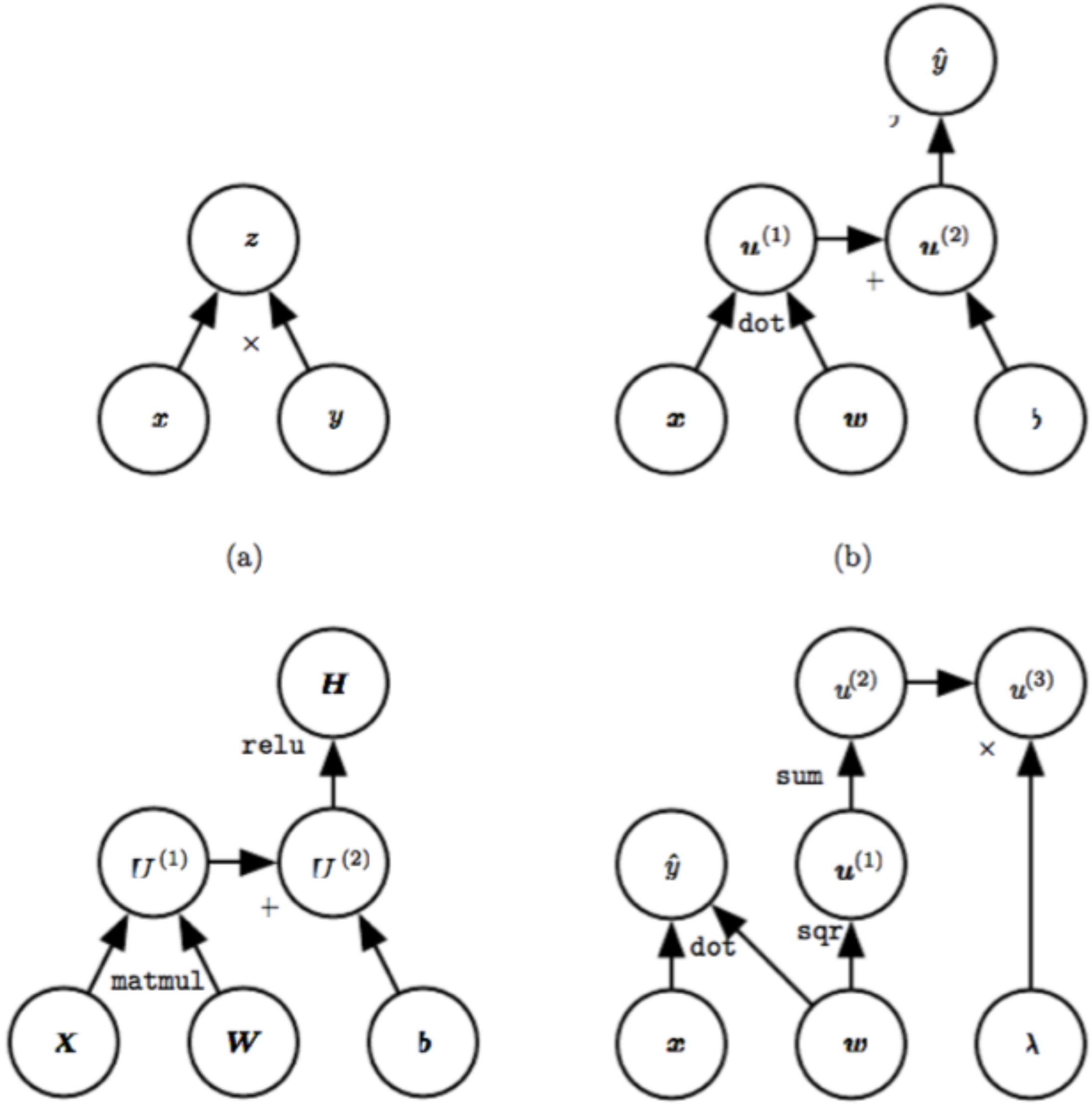
From single weight W_{ij} to full W:

$$\frac{\partial s}{\partial W_{ij}} = \delta_i x_j$$

- We want all combinations of $i = 1, 2, \ldots$ and $j = 1, 2, 3, \ldots$
- Solution: Outer product

$$\frac{\partial J}{\partial W} = \delta x^T$$

Computation Graphs

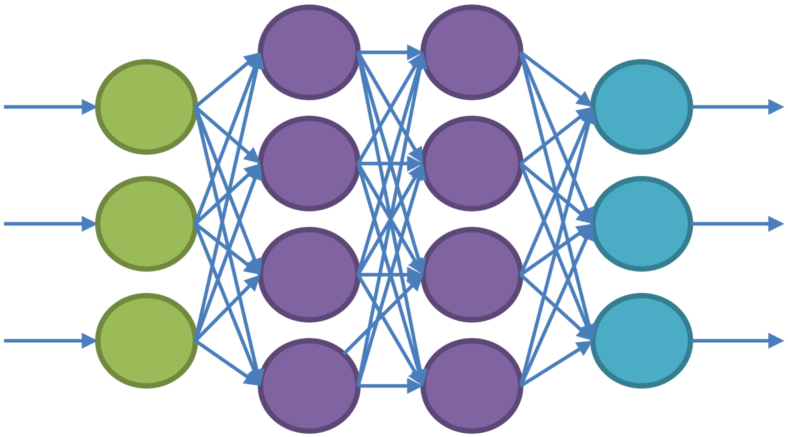


BACKPROPAGATION

Slides from Intel

How to Train a Neural Net?

Input
(Feature Vector)



Output
(Label)

- Put in Training inputs, get the output
- Compare output to correct answers: Look at loss function J
- Adjust and repeat!
- Backpropagation tells us how to make a single adjustment using calculus.

How have we trained before?

- Gradient Descent!

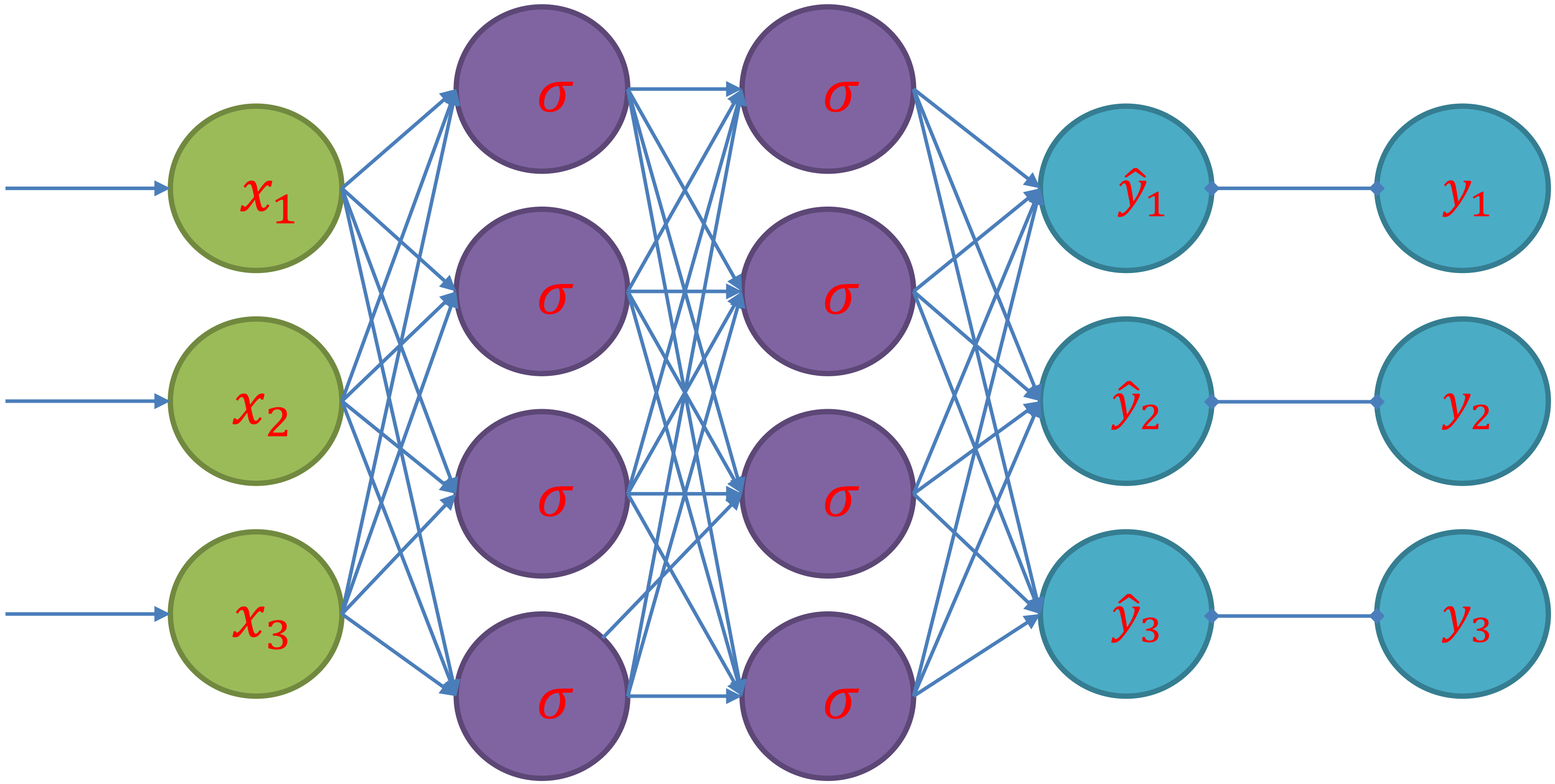
1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

How have we trained before?

- Gradient Descent!

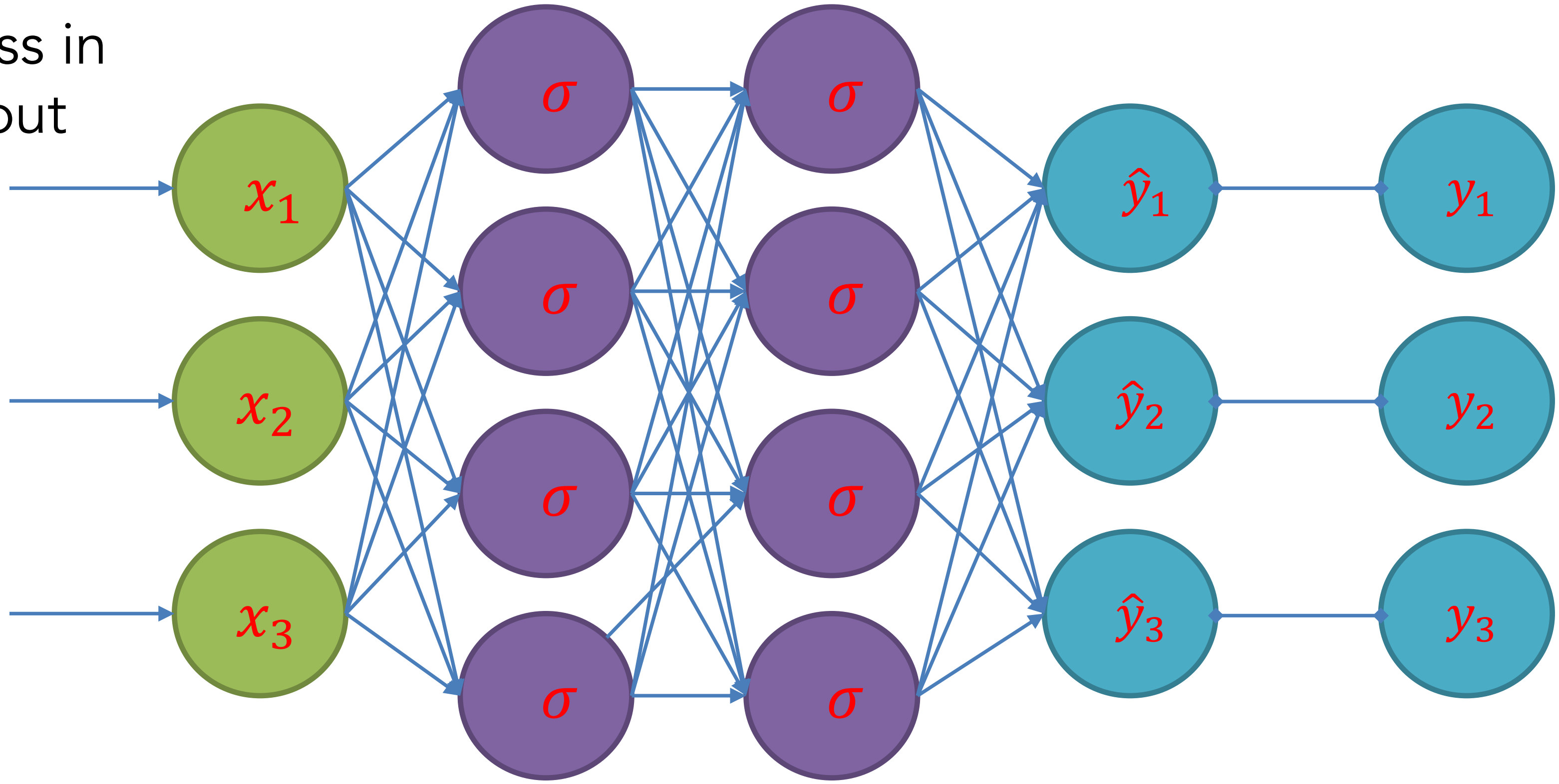
1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

Feedforward Neural Network



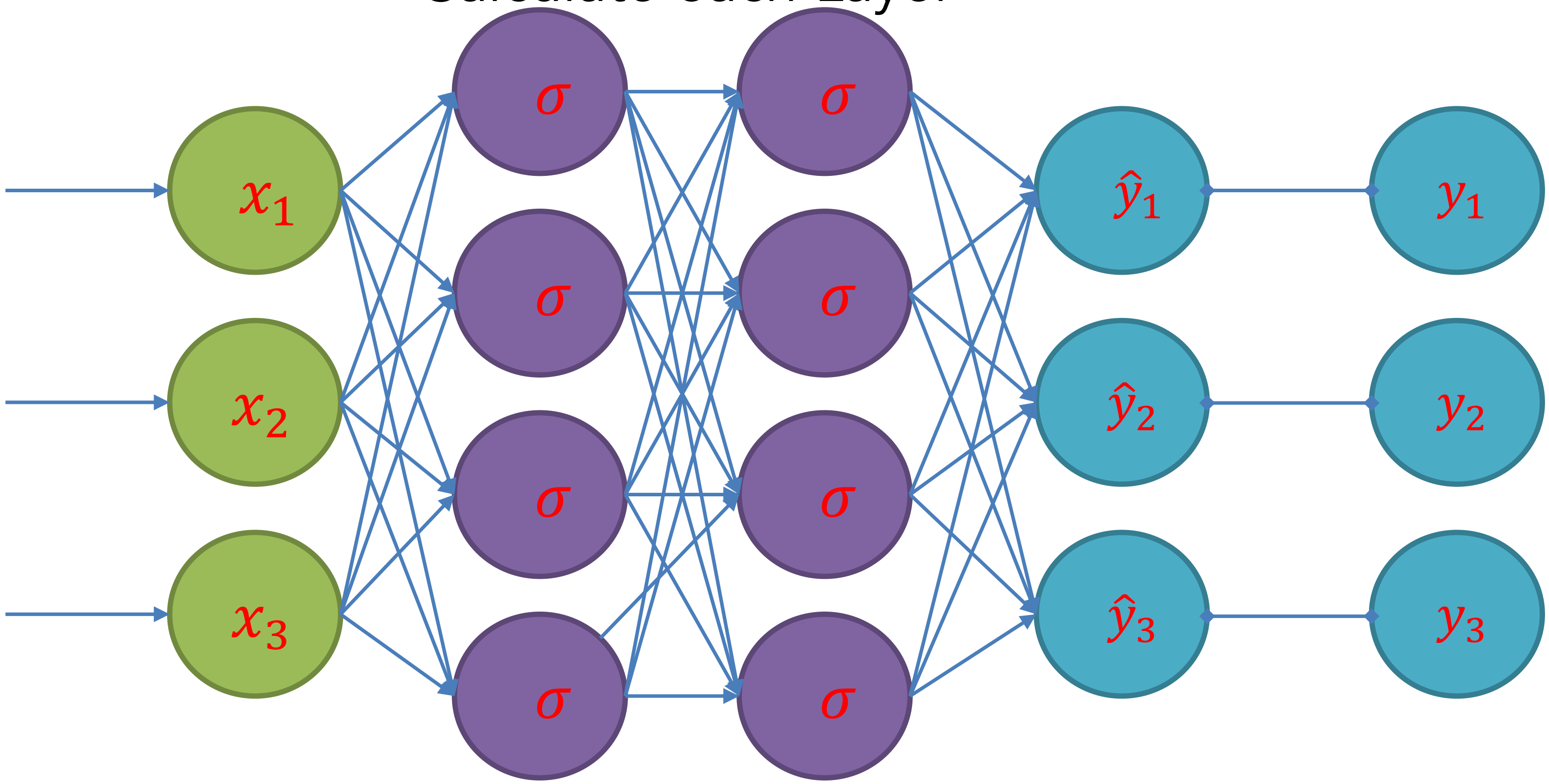
Forward Propagation

Pass in
Input

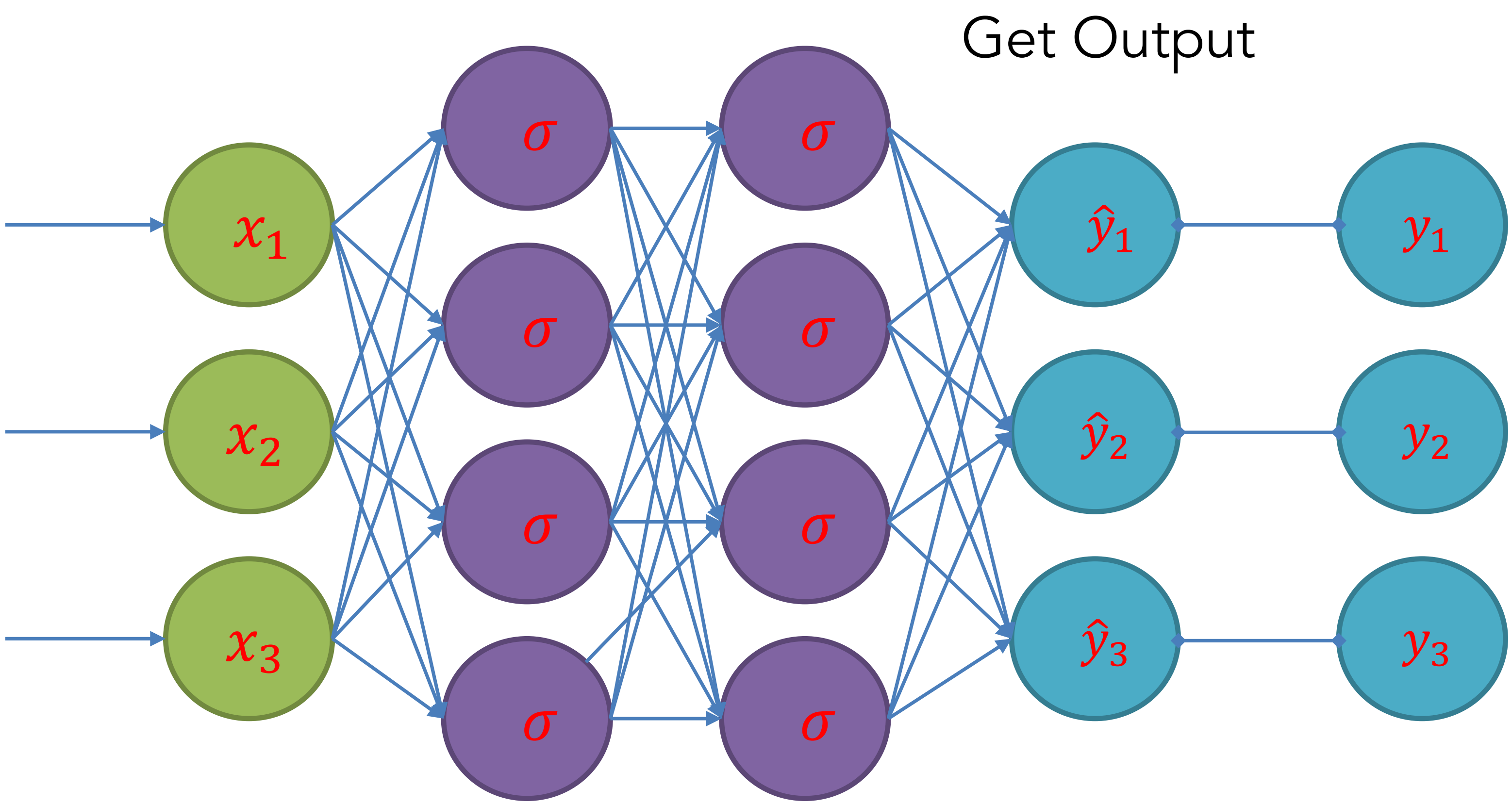


Forward Propagation

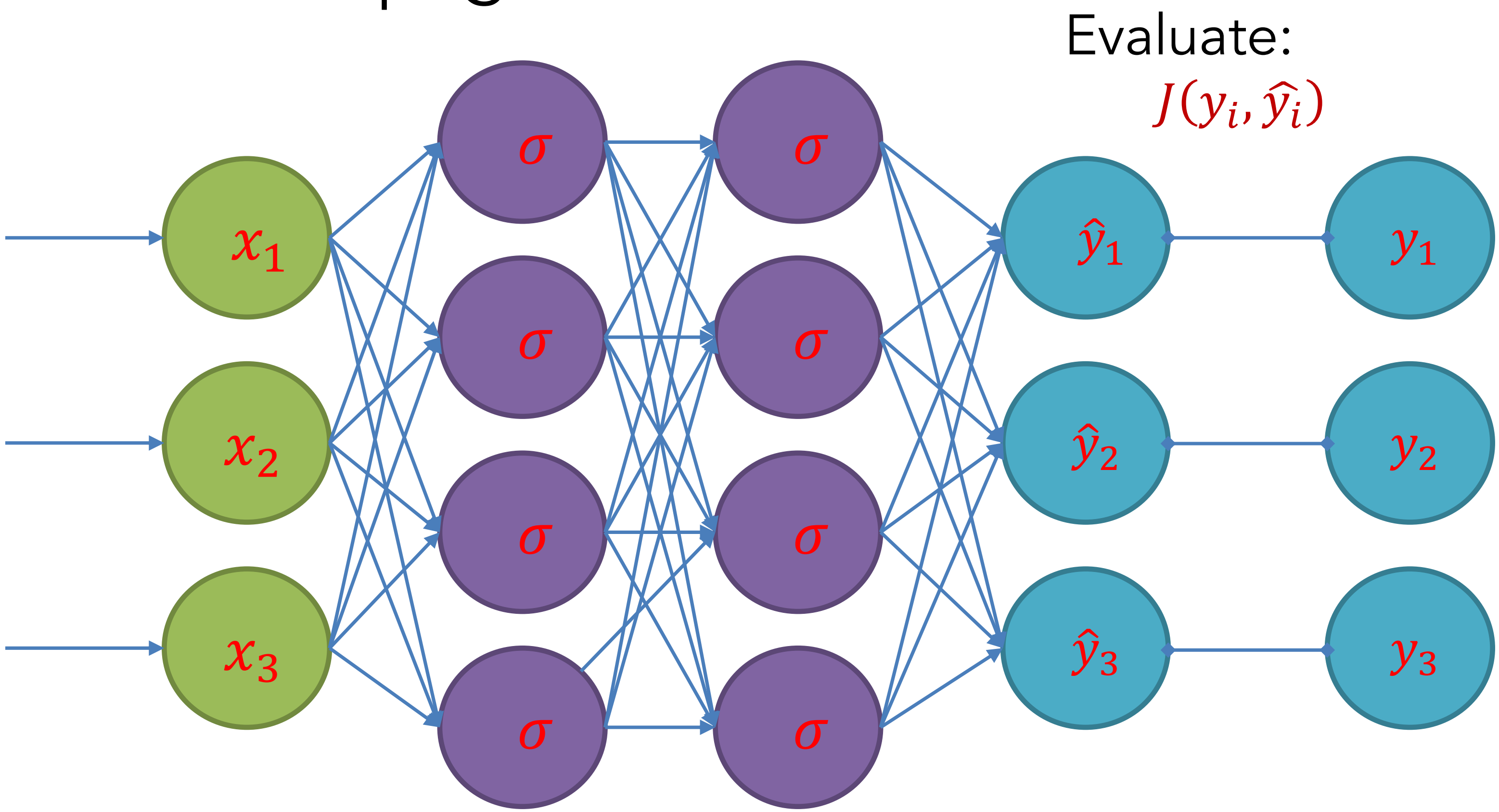
Calculate each Layer



Forward Propagation



Forward Propagation



How have we trained before?

- Gradient Descent!

1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

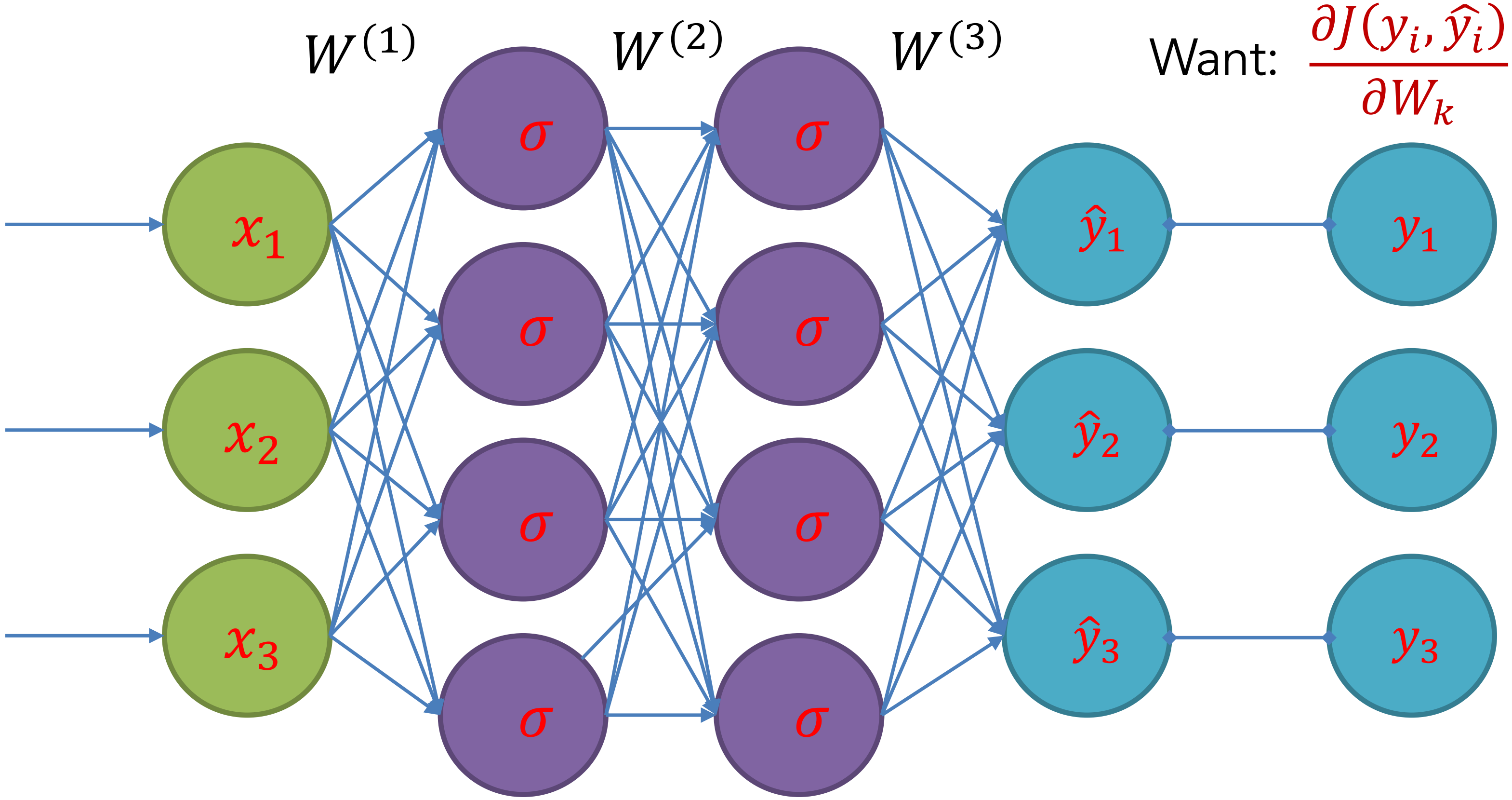
How to Train a Neural Net?

- How could we change the weights to make our Loss Function lower?
- Think of neural net as a function F: X -> Y
- F is a complex computation involving many weights W_k
- Given the structure, the weights “define” the function F (and therefore define our model)
- Loss Function is J(y,F(x))

How to Train a Neural Net?

- Get $\frac{\partial J}{\partial W_k}$ for every weight in the network.
- This tells us what direction to adjust each W_k if we want to lower our loss function.
- Make an adjustment and repeat!

Feedforward Neural Network



Calculus to the Rescue

- Use calculus, chain rule, etc. etc.
- Functions are chosen to have “nice” derivatives
- Numerical issues to be considered

Punchline

$$\frac{\partial J}{\partial W^{(3)}} = (\hat{y} - y) \cdot a^{(3)}$$

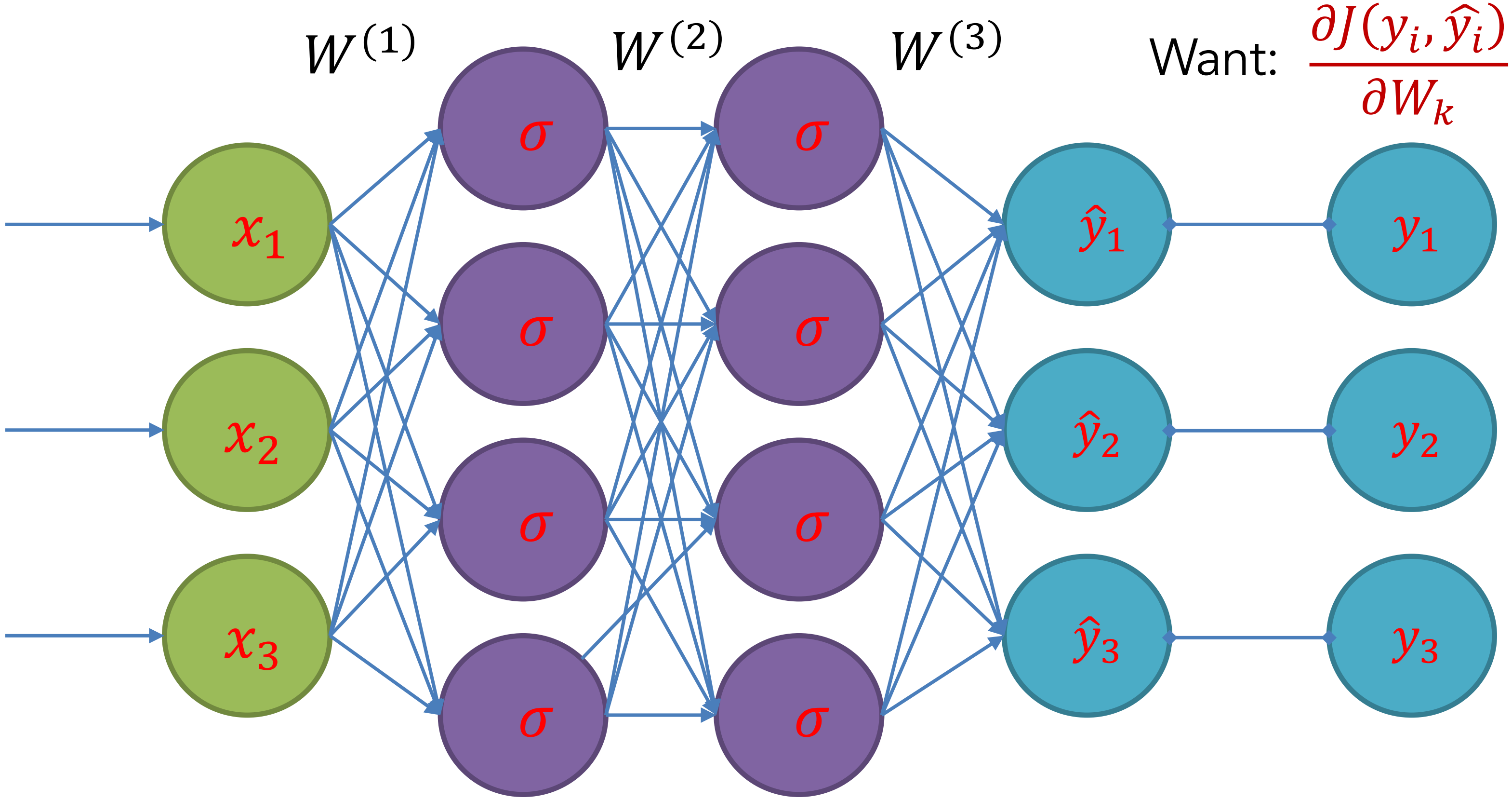
$$\frac{\partial J}{\partial W^{(2)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot a^{(2)}$$

$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot W^{(2)} \cdot \sigma'\left(z^{(2)}\right) \cdot X$$

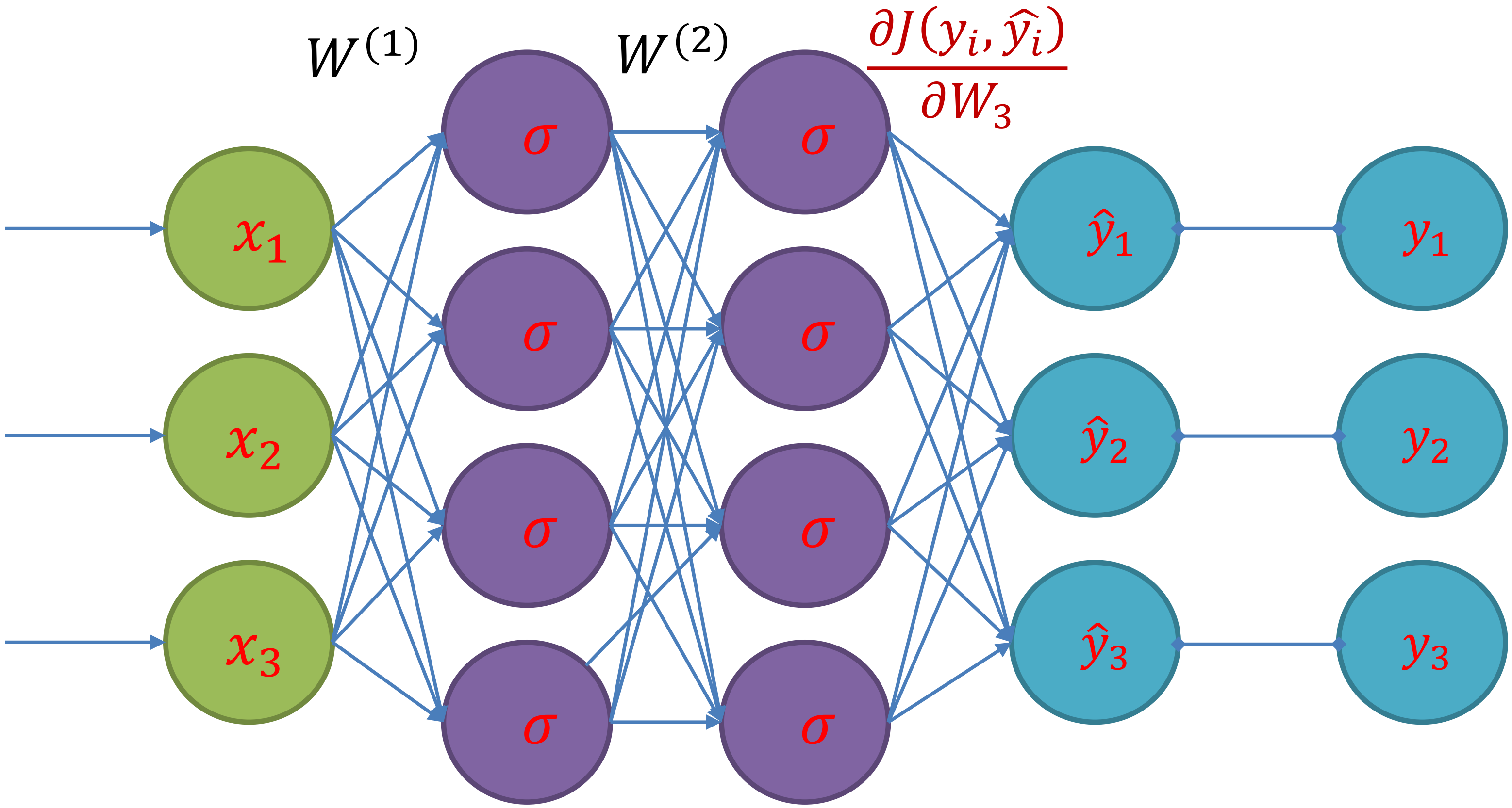
- Recall that: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- Though they appear complex, above are easy to compute!

Backpropagation

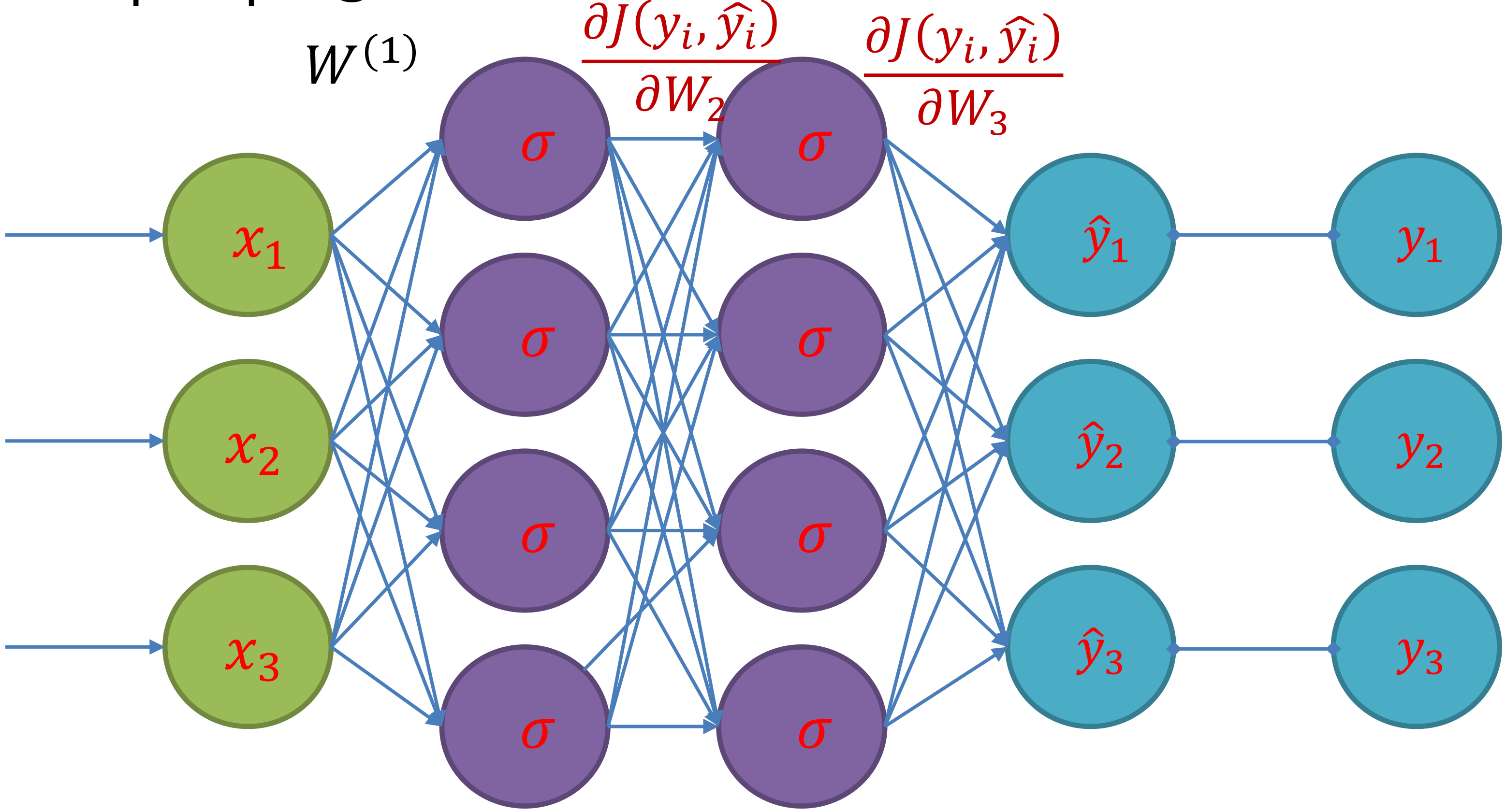
Want: $\frac{\partial J(y_i, \hat{y_i})}{\partial W_k}$



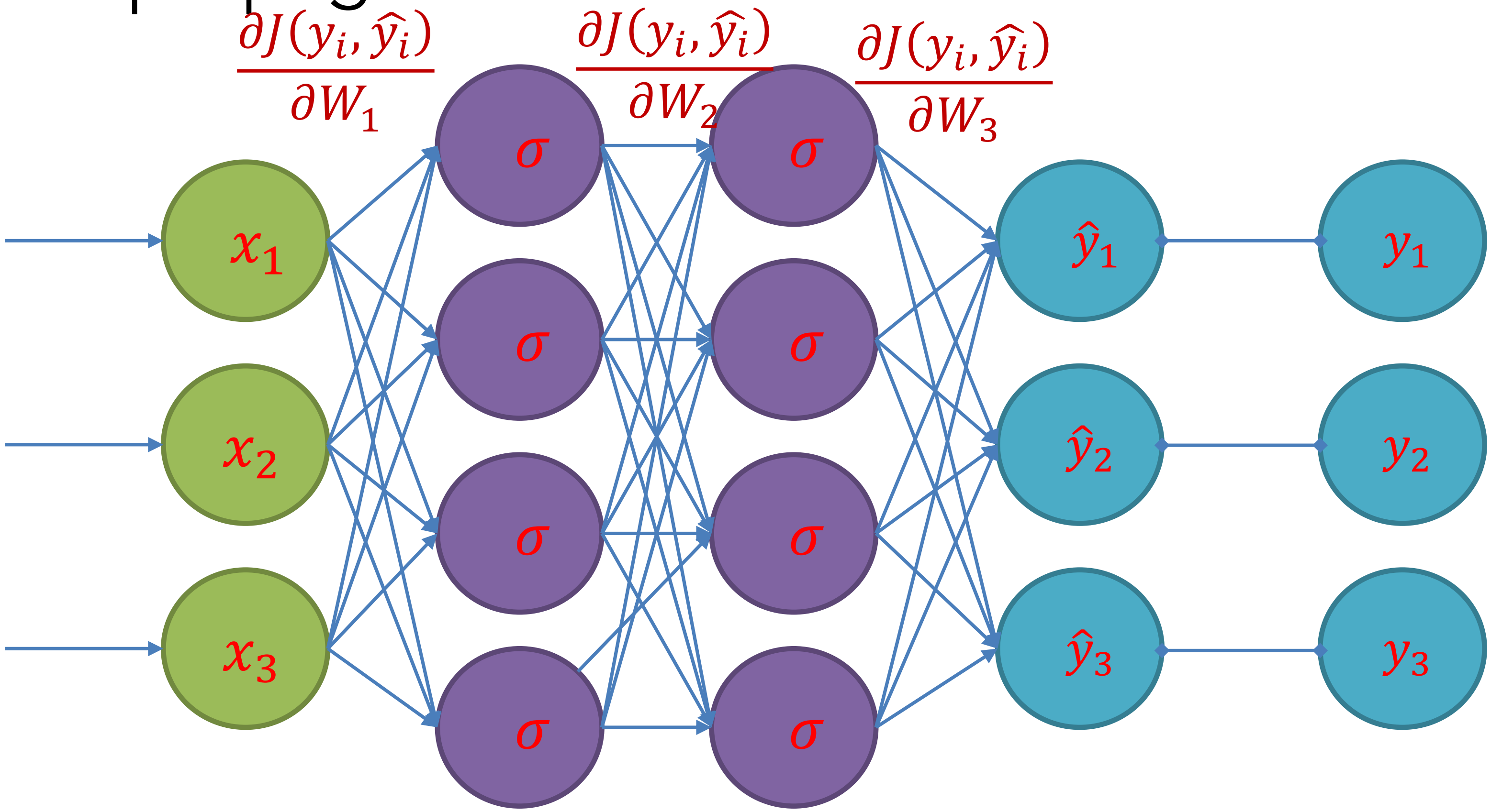
Backpropagation



Backpropagation



Backpropagation



How have we trained before?

- Gradient Descent!

1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

Vanishing Gradients

Recall that:

$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot W^{(2)} \cdot \sigma'\left(z^{(2)}\right) \cdot X$$

- Remember: $\sigma'(z) = \sigma(z)\left(1 - \sigma(z)\right) \leq .25$
- As we have more layers, the gradient gets very small at the early layers.
- This is known as the "vanishing gradient" problem.
- For this reason, other activations (such as ReLU) have become more common.

Neural Networks – What we learnt

- Neural networks for supervised learning
- Multiple Hidden layers as universal approximators
- Fully connected feed-forward networks for calssification
- Backpropagation for learning network parameters (Weights at layers)

Up Next:

- Paradigms for Deep Learning Models
  - CNNs
  - RNNs

CONVOLUTIONAL NEURAL NETWORKS

Motivation – Image Data

- So far, the structure of our neural network treats all inputs interchangeably.
- No relationships between the individual inputs
- Just an ordered set of variables

- We want to incorporate domain knowledge into the architecture of a Neural Network.

Motivation

- Image data has important structures, such as;

- "Topology" of pixels
- Translation invariance
- Issues of lighting and contrast
- Knowledge of human visual system
- Nearby pixels tend to have similar values
- Edges and shapes
- Scale Invariance – objects may appear at different sizes in the image.

Motivation – Image Data

- Fully connected would require a vast number of parameters
- MNIST images are small (32 x 32 pixels) and in grayscale
- Color images are more typically at least (200 x 200) pixels x 3 color channels (RGB) = 120,000 values.
- A single fully connected layer would require $(200 \times 200 \times 3)^2$ = 14,400,000,000 weights!
- Variance (in terms of bias-variance) would be too high
- So we introduce “bias” by structuring the network to look for certain kinds of patterns

Motivation

- Features need to be “built up”
- Edges -> shapes -> relations between shapes
- Textures

- Cat = two eyes in certain relation to one another + cat fur texture.
- Eyes = dark circle (pupil) inside another circle.
- Circle = particular combination of edge detectors.
- Fur = edges in certain pattern.

Kernels

- A *kernel* is a grid of weights “overlaid” on image, centered on one pixel
- Each weight multiplied with pixel underneath it
- Output over the centered pixel is $\sum_{p=1}^{P} W_p \cdot pixel_p$
- Used for traditional image processing techniques:
  - Blur
  - Sharpen
  - Edge detection
  - Emboss

Kernel: 3x3 Example

Input

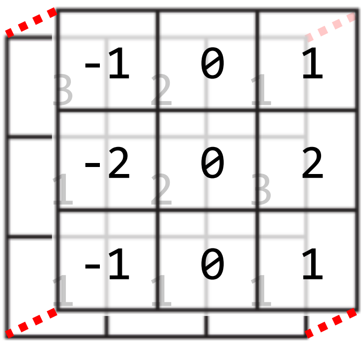
3	2	1
1	2	3
1	1	1

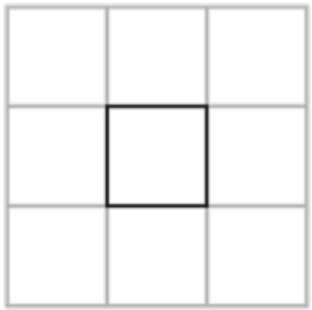
Kernel

-1	0	1
-2	0	2
-1	0	1

Output

Kernel: 3x3 Example





Kernel: 3x3 Example

Input

3	2	1
1	2	3
1	1	1

Kernel

-1	0	1
-2	0	2
-1	0	1

Output

	2	

$$
\begin{aligned}
&= (3 \cdot -1) + (2 \cdot 0) + (1 \cdot 1) \\
&+(1 \cdot -2) + (2 \cdot 0) + (3 \cdot 2) \\
&+(1 \cdot -1) + (1 \cdot 0) + (1 \cdot 1)
\end{aligned}
$$

$$= -3 + 1 - 2 + 6 - 1 + 1 = 2$$

Kernels as Feature Detectors

Can think of kernels as a "local feature detectors"

Vertical Line Detector

-1	1	-1
-1	1	-1
-1	1	-1

Horizontal Line Detector

-1	-1	-1
1	1	1
-1	-1	-1

Corner Detector

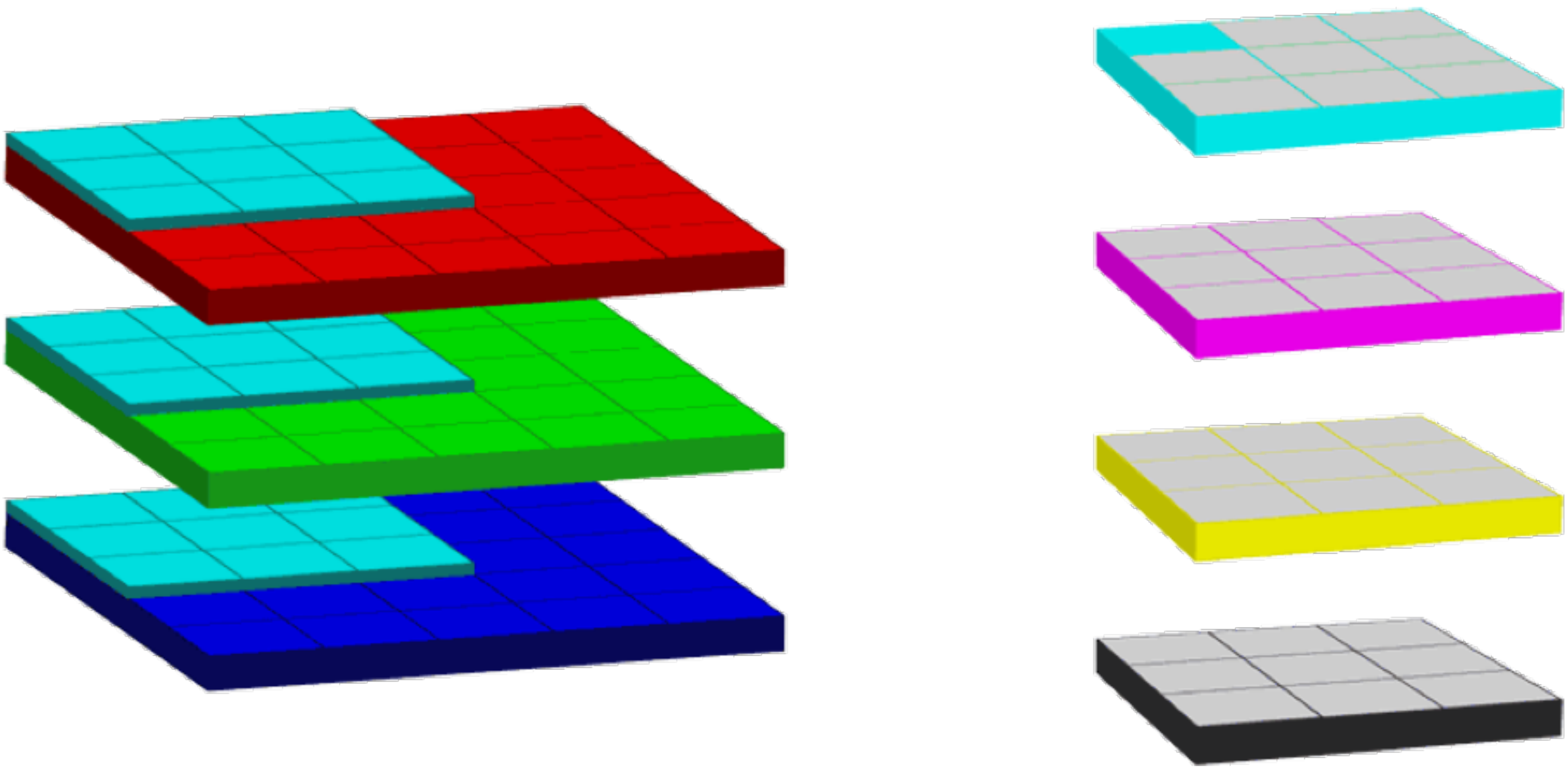
-1	-1	-1
-1	1	1
-1	1	1

Convolutional Neural Nets

Primary Ideas behind Convolutional Neural Networks:

- Let the Neural Network learn which kernels are most useful
- Use same set of kernels across entire image (translation invariance)
- Reduces number of parameters and “variance” (from bias-variance point of view)

Convolutions

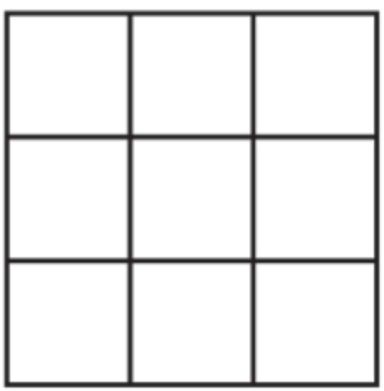


Convolution Settings – Grid Size

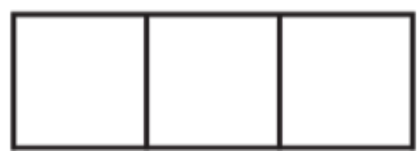
Grid Size (Height and Width):

- The number of pixels a kernel "sees" at once
- Typically use odd numbers so that there is a "center" pixel
- Kernel does not need to be square

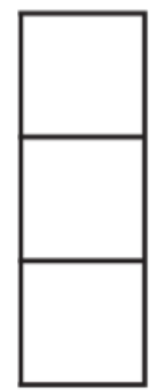
Height: 3, Width: 3



Height: 1, Width: 3



Height: 3, Width: 1

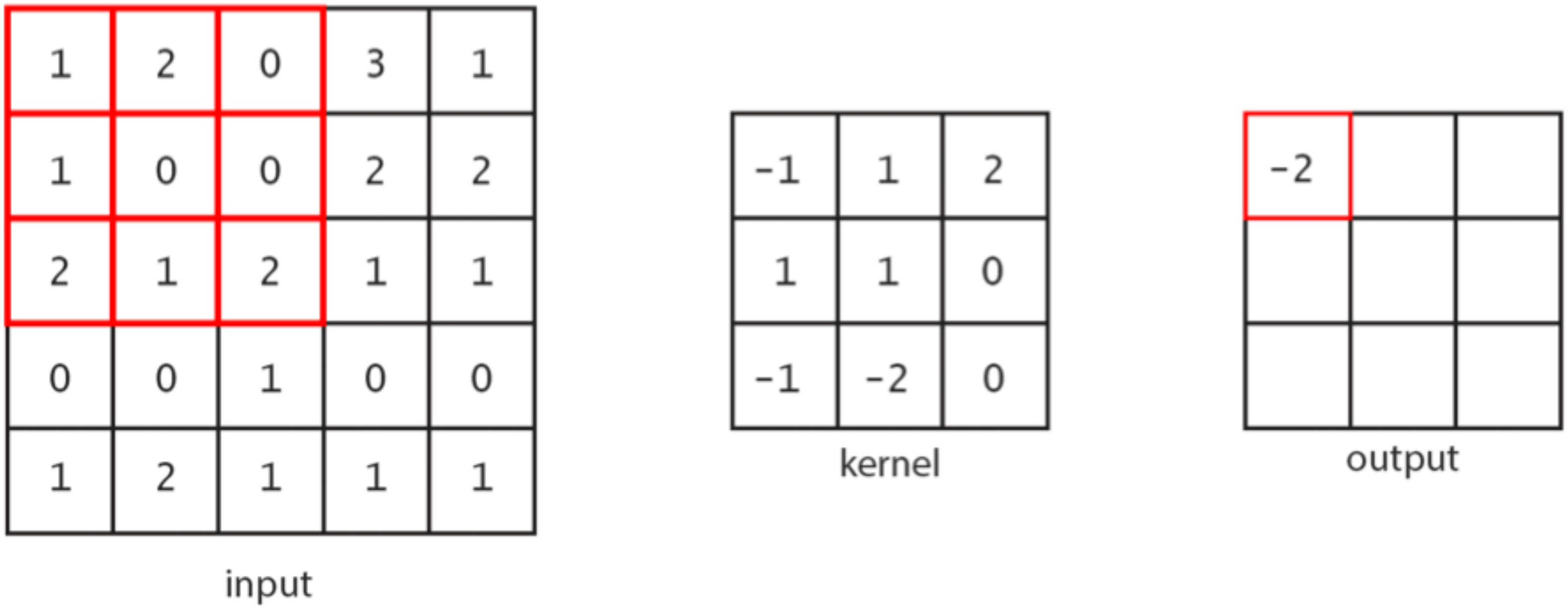


Convolution Settings - Padding

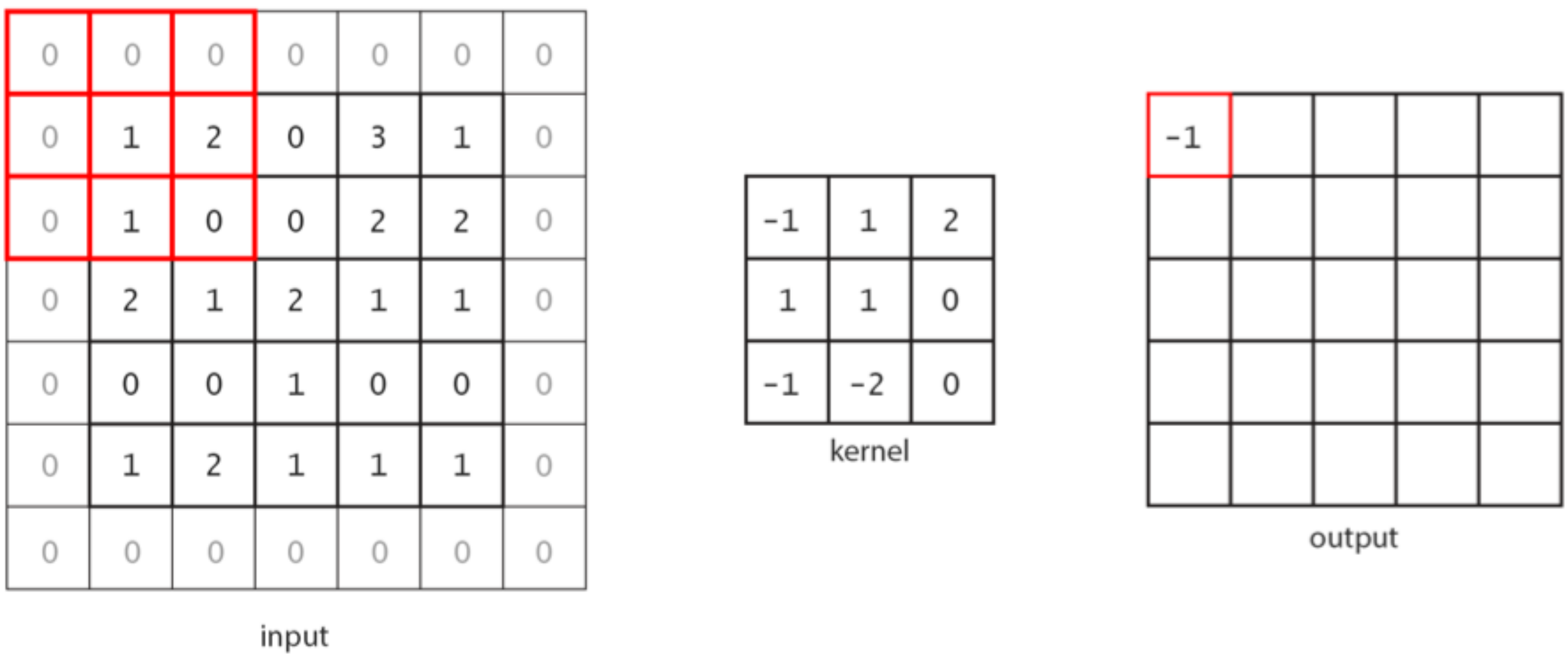
Padding

- Using Kernels directly, there will be an “edge effect”
- Pixels near the edge will not be used as “center pixels” since there are not enough surrounding pixels
- Padding adds extra pixels around the frame
- So every pixel of the original image will be a center pixel as the kernel moves across the image
- Added pixels are typically of value zero (zero-padding)

Without Padding



With Padding

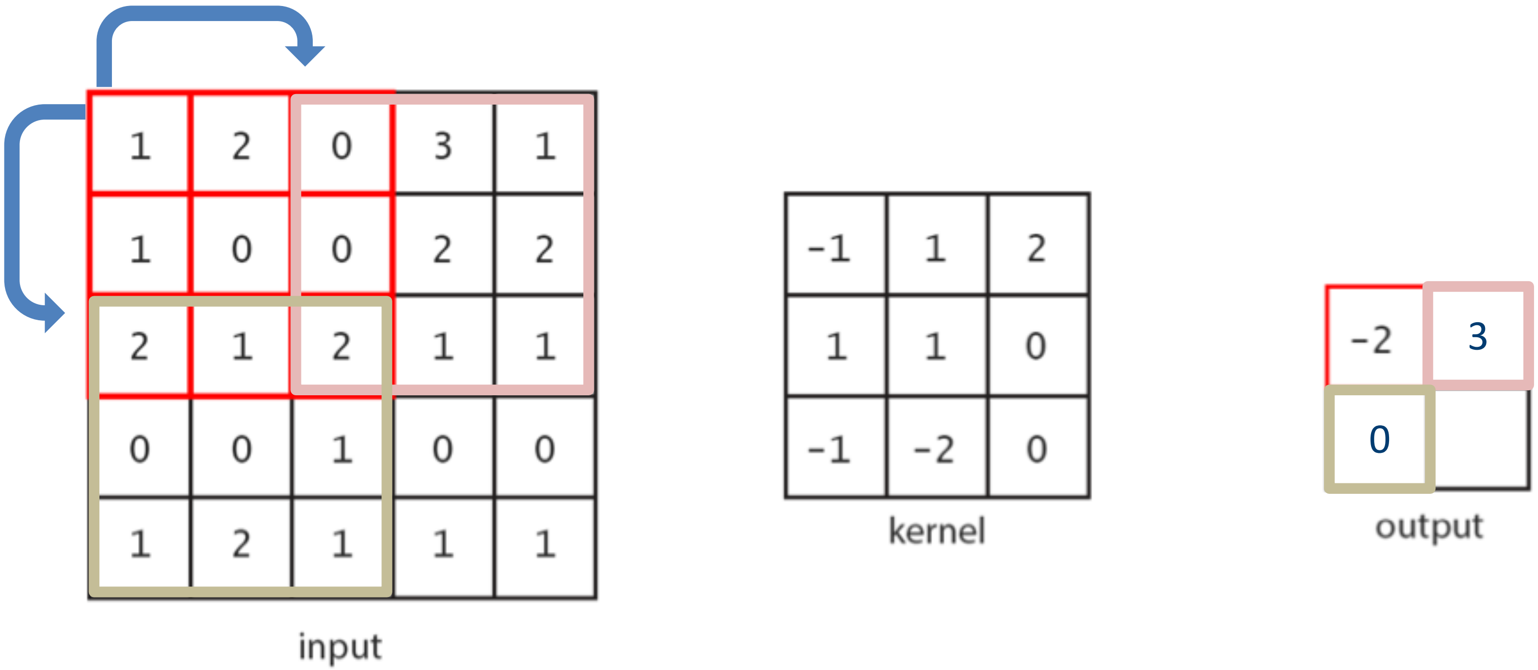


Convolution Settings

Stride

- The "step size" as the kernel moves across the image
- Can be different for vertical and horizontal steps (but usually is the same value)
- When stride is greater than 1, it scales down the output dimension

Stride 2 Example – No Padding



Stride 2 Example – With Padding

0	0	0	0	0	0	0
0	1	2	0	3	1	0
0	1	0	0	2	2	0
0	2	1	2	1	1	0
0	0	0	1	0	0	0
0	1	2	1	1	1	0
0	0	0	0	0	0	0

input

-1	1	2
1	1	0
-1	-2	0

kernel

-1	2	
3		

output

Convolutional Settings - Depth

- In images, we often have multiple numbers associated with each pixel location.
- These numbers are referred to as "channels"
  - RGB image – 3 channels
  - CMYK – 4 channels
- The number of channels is referred to as the "depth"
- So the kernel itself will have a "depth" the same size as the number of input channels
- Example: a 5x5 kernel on an RGB image
  - There will be 5x5x3 = 75 weights

Convolutional Settings - Depth

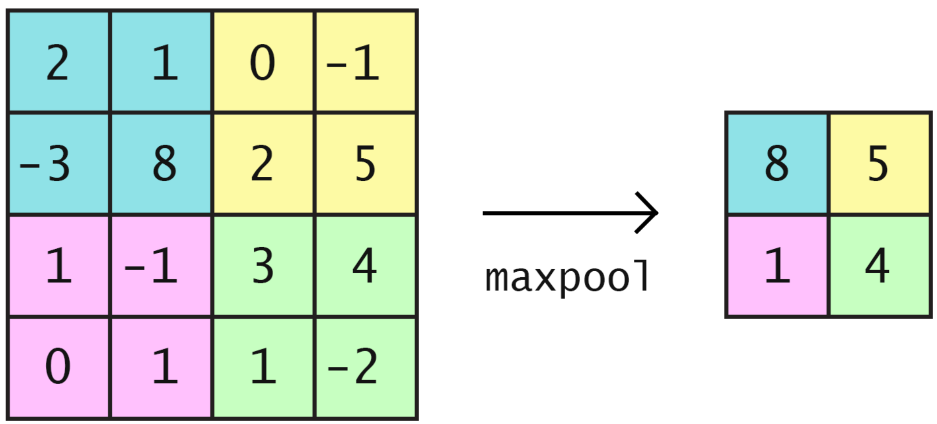
- The output from the layer will also have a depth
- The networks typically train many different kernels
- Each kernel outputs a single number at each pixel location
- So if there are 10 kernels in a layer, the output of that layer will have depth 10.

Pooling

- Idea: Reduce the image size by mapping a patch of pixels to a single value.
- Shrinks the dimensions of the image.
- Does not have parameters, though there are different types of pooling operations.

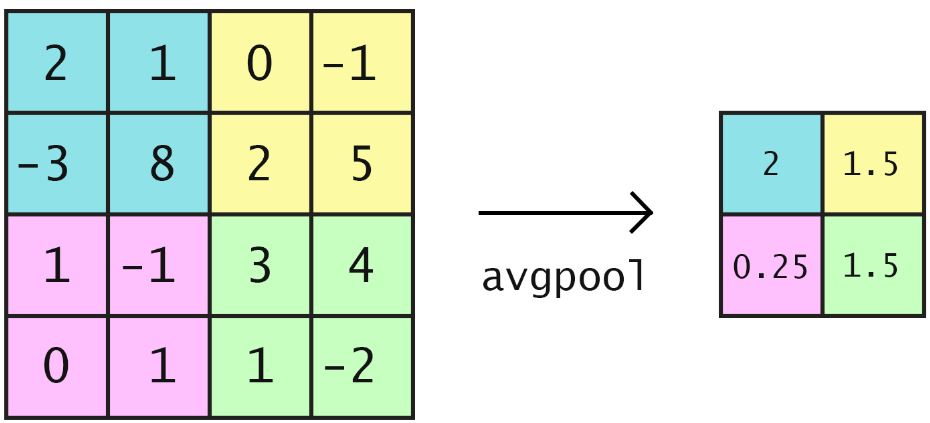
Pooling: Max-pool

- For each distinct patch, represent it by the maximum
- 2x2 maxpool shown below

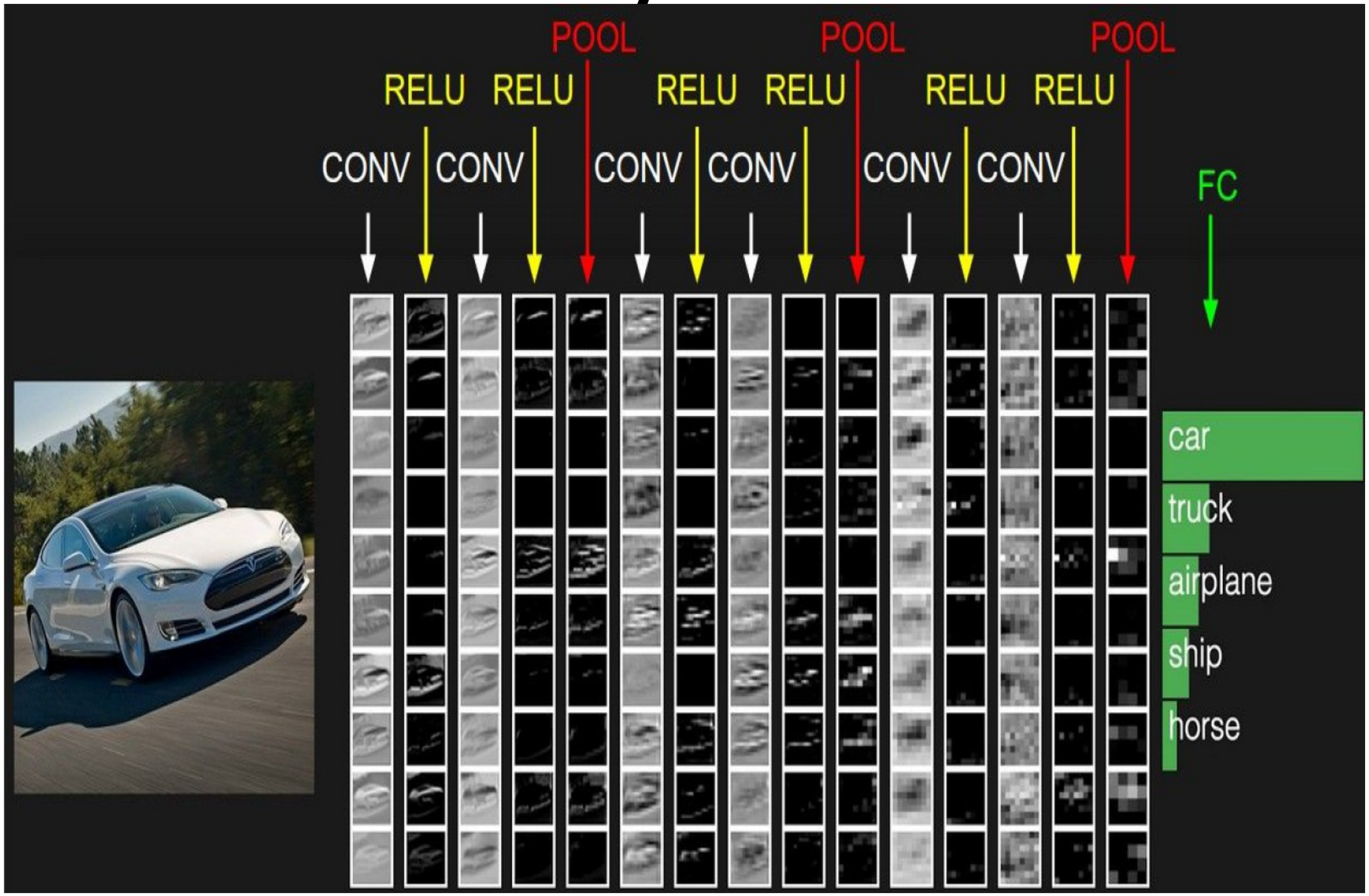


Pooling: Average-pool

- For each distinct patch, represent it by the average
- 2x2 avgpool shown below.

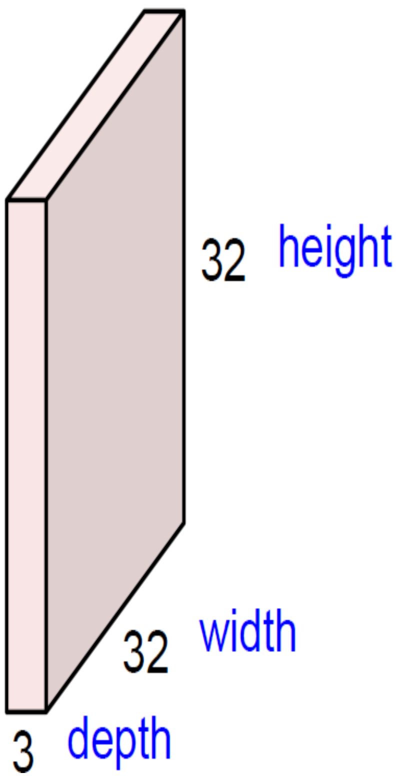


ConvNet: CONV, RELU, POOL and FC Layers



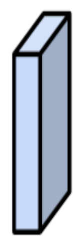
Convolution Layer

32x32x3 image



Filters always extend the full depth of the input volume

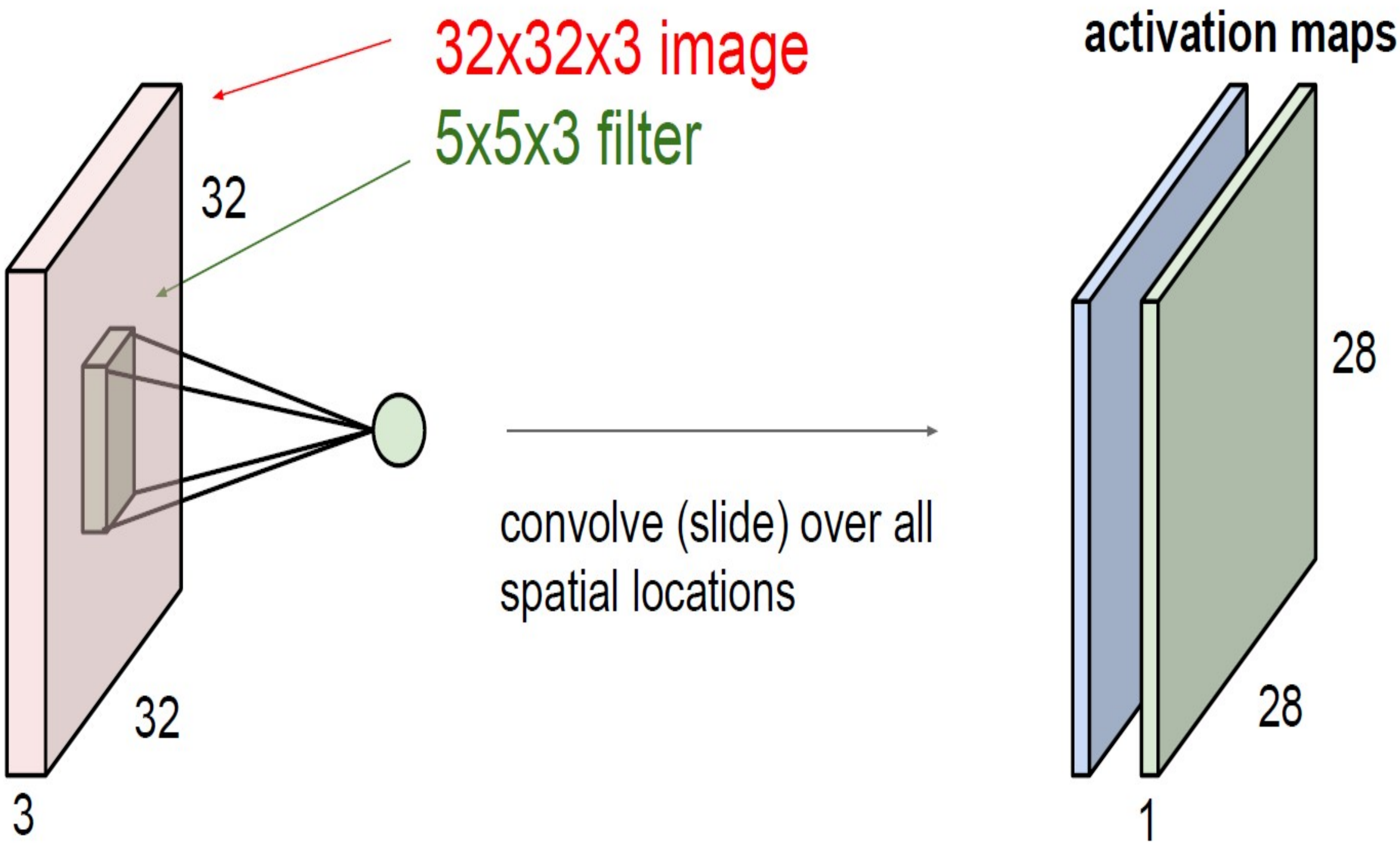
5x5x3 filter



Convolve the filter with the image
i.e. “slide over the image spatially, computing dot products”

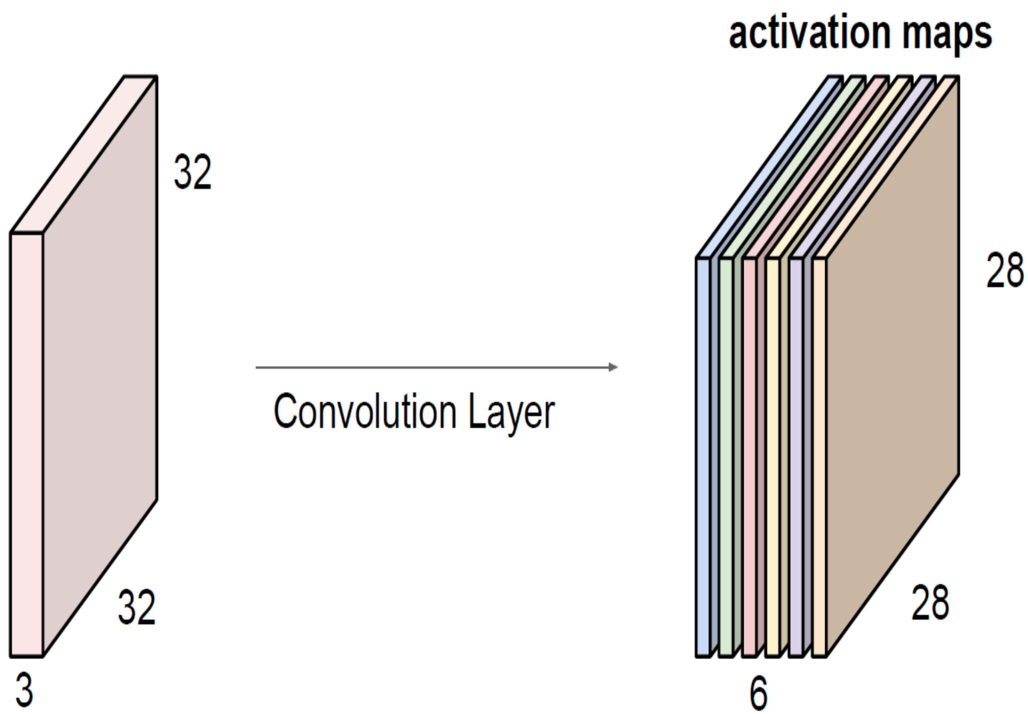
Convolution Layer

consider a second, green filter



Convolution Layer

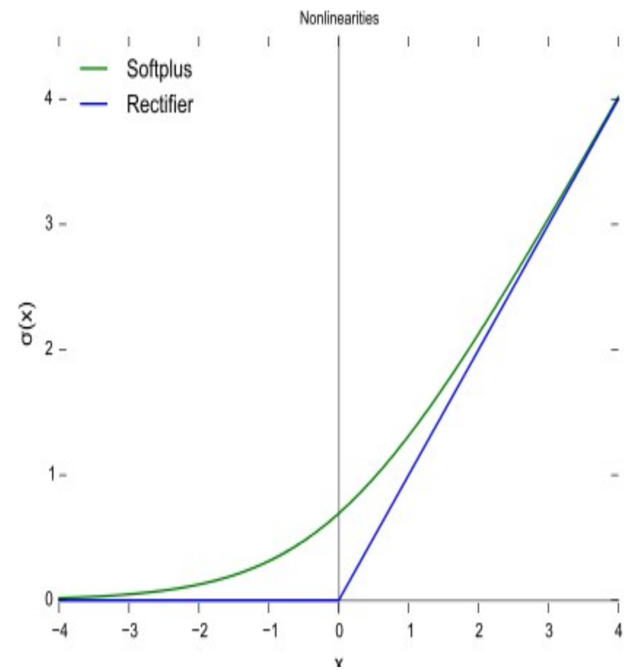
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

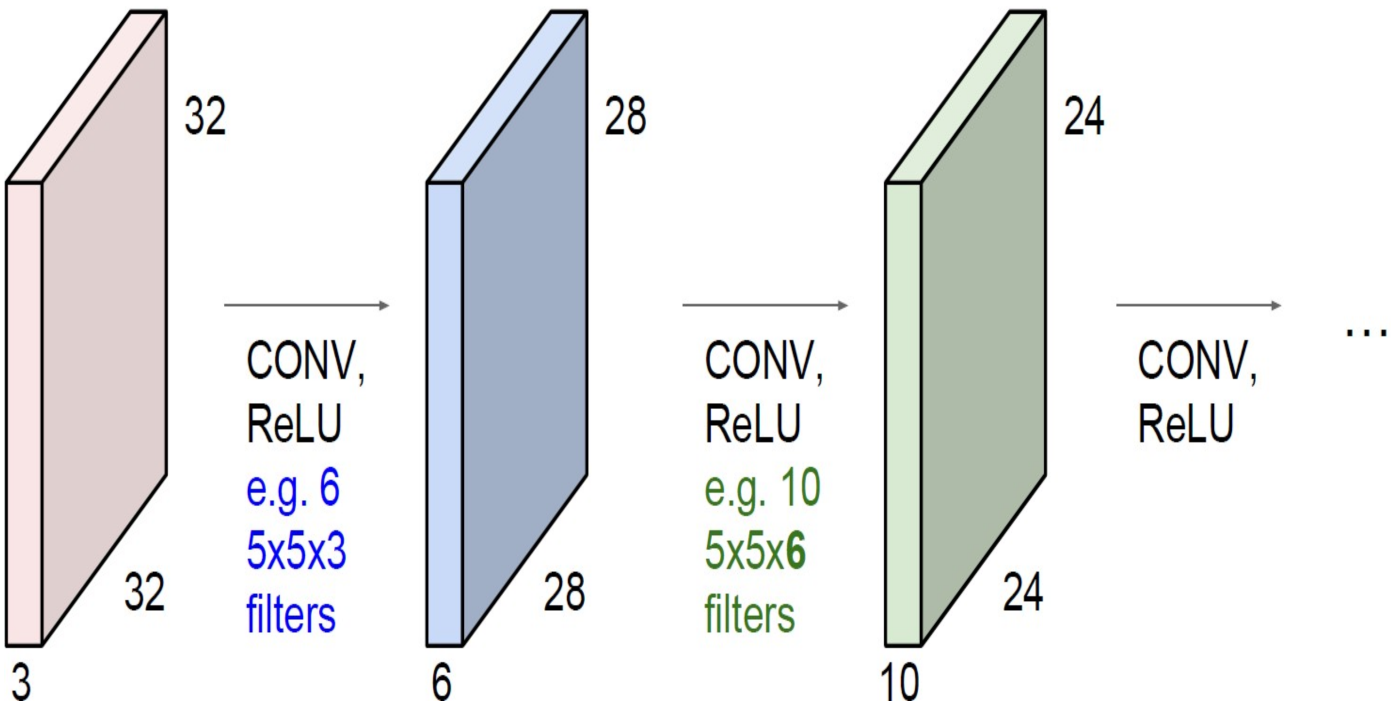
ReLU (Rectified Linear Units) Layer

- This is a layer of neurons that applies the activation function $f(x)=\max(0,x)$.
- It increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer.
- Other functions are also used to increase nonlinearity, for example the hyperbolic tangent $f(x)=\tanh(x)$, and the sigmoid function.
- This is also known as a ramp function.

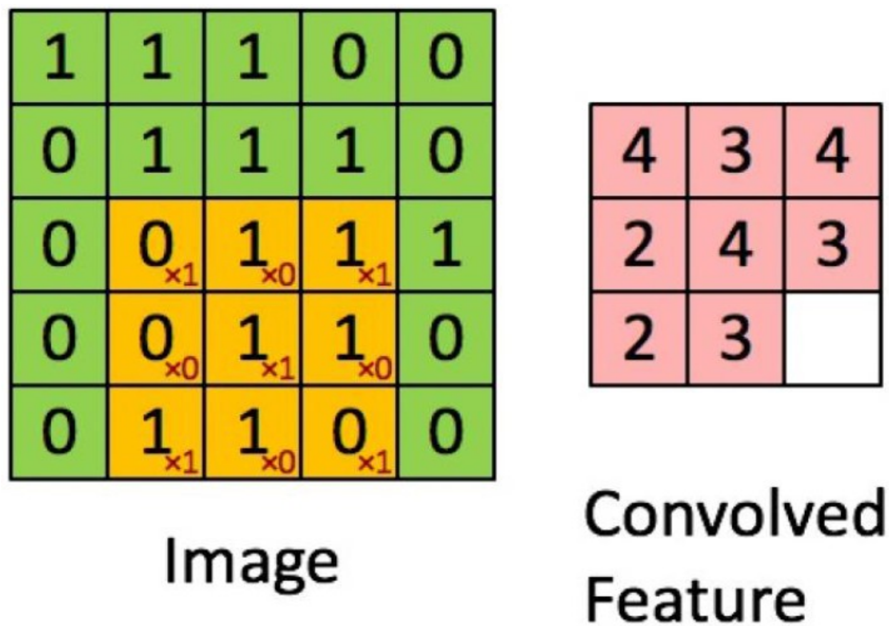


A Basic ConvNet

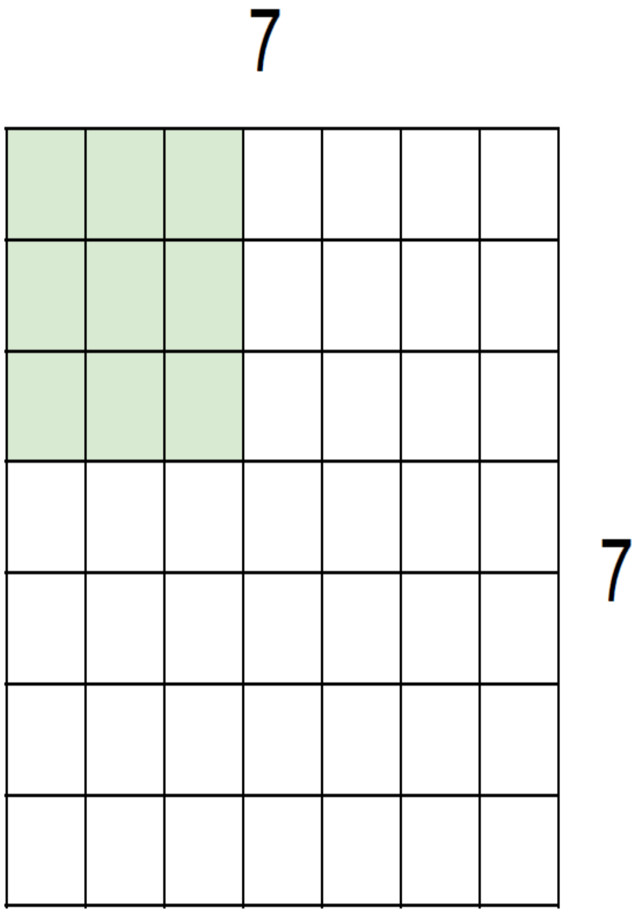
Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



What is convolution of an image with a filter



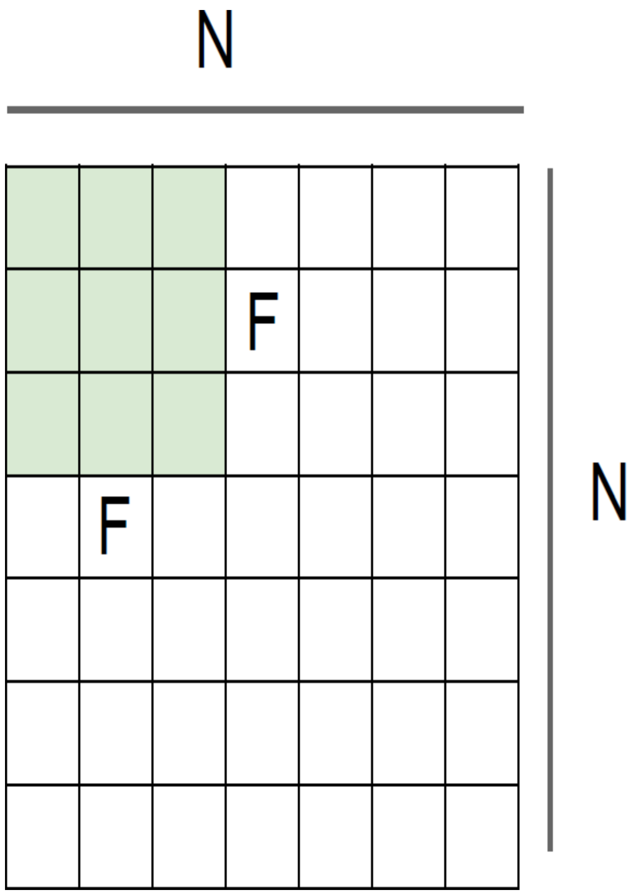
Details about the convolution layer



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.

Details about the convolution layer

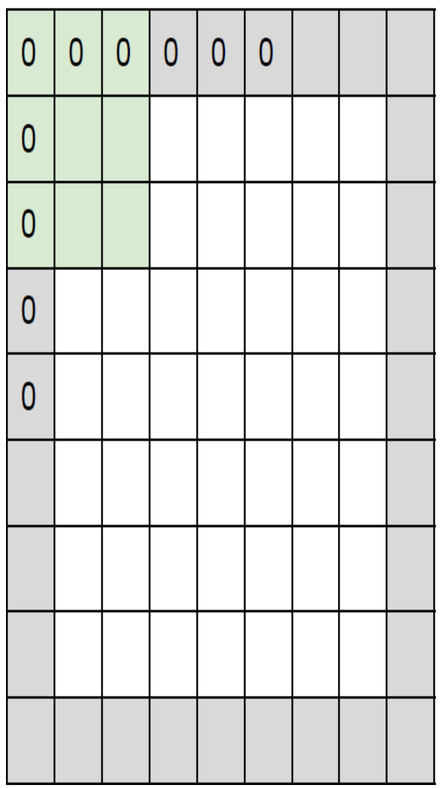


Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33

Details about the convolution layer

In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Convolution layer examples

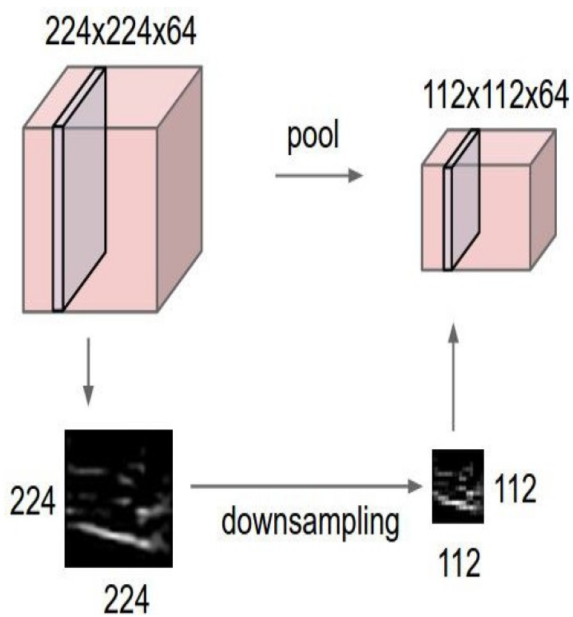
Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

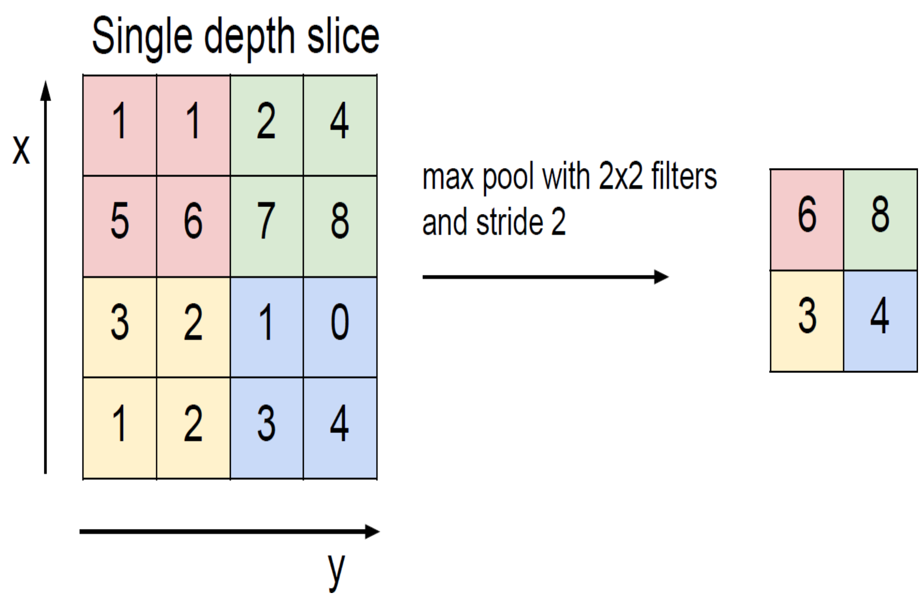
Output volume size: ?

(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

Pooling Layer

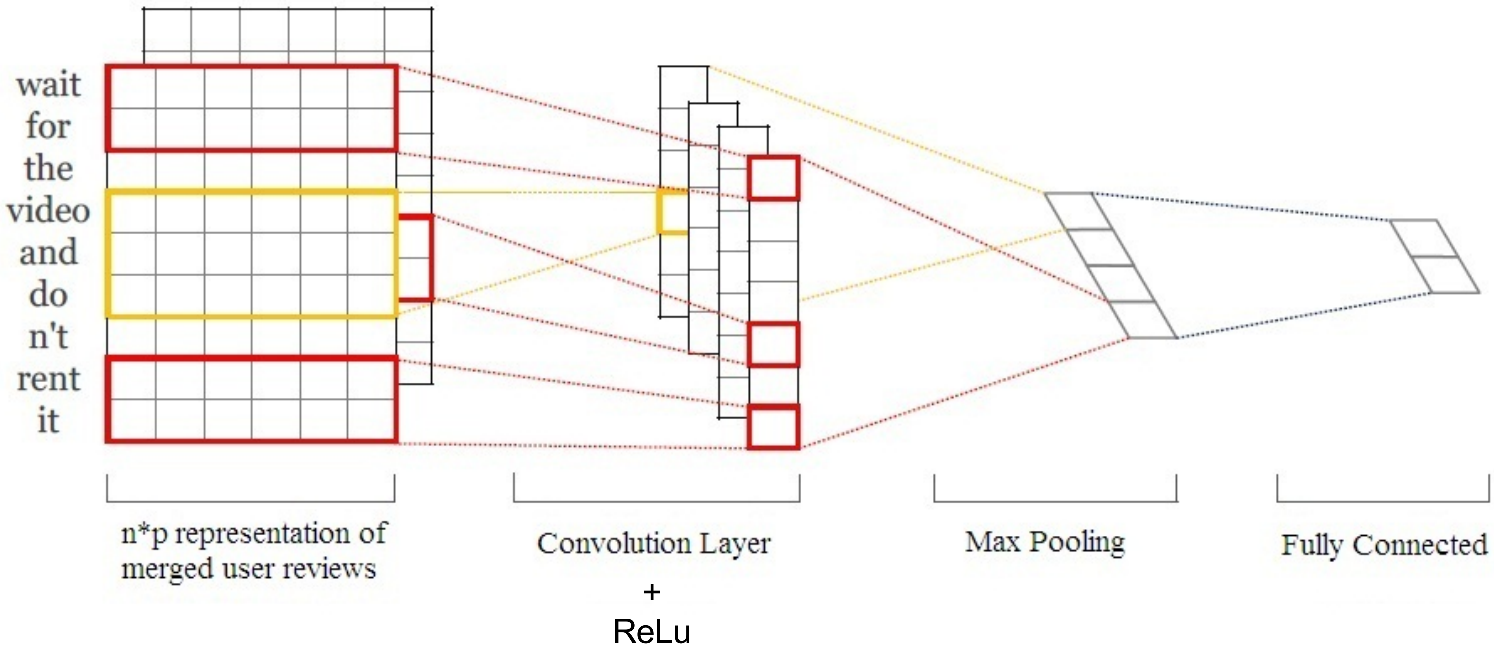
makes the representations smaller and more manageable
operates over each activation map independently:





- Invariance to image transformation and increases compactness to representation.
- Pooling types: Max, Average, L2 etc.

Convolutional Neural Networks

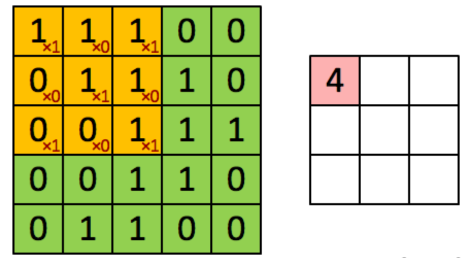


$$z_j = f(V^u_{1:n} * K_j + b_j)$$

Where ReLu is used as f.

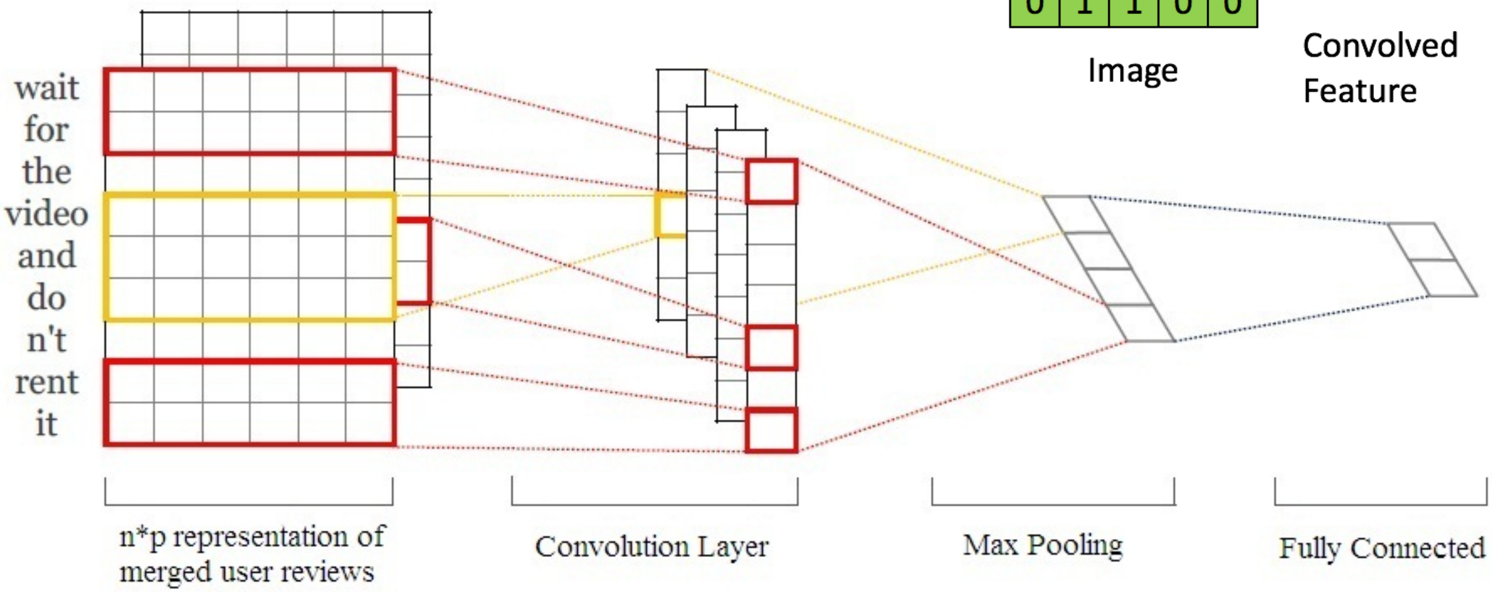
Convolutional Neural Networks

Kernel= $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$



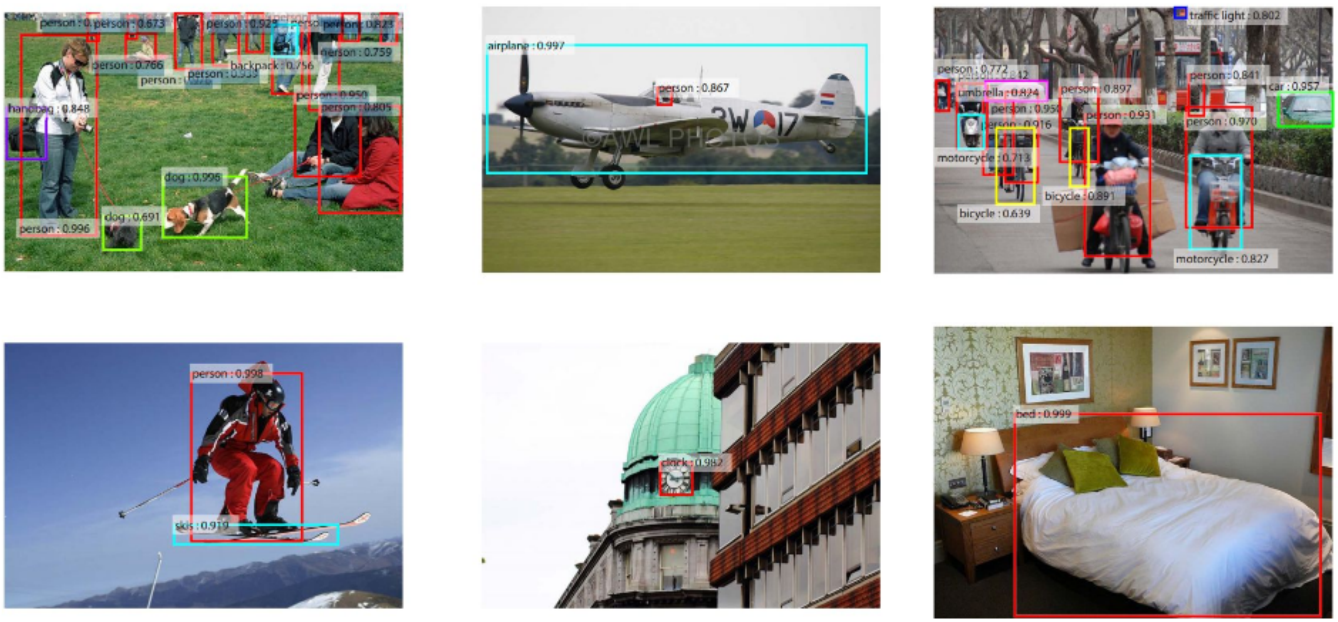
Image

Convolved Feature



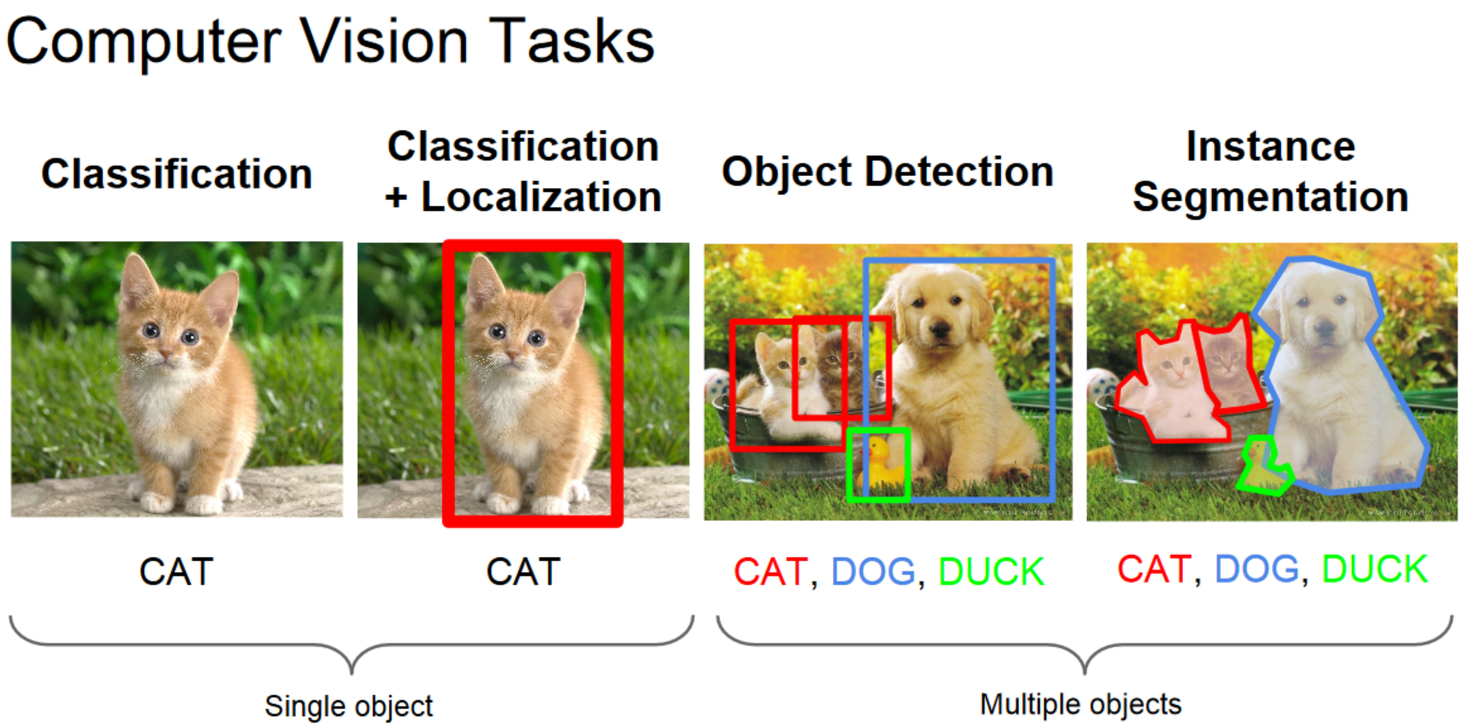
Applications

Localization and Detection



Results from Faster R-CNN, Ren et al 2015

Applications



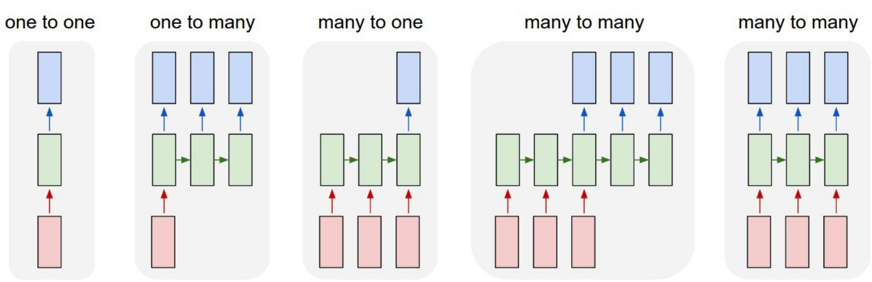
Is deep learning all about CNNs?

- Consider a language modelling task
- Given a vocabulary, the task is to predict the next word in a sentence
- Sequence information of words are important
- Typically in cases where sequential data is involved, **recurrent neural networks (RNNs)** are widely used

Recurrent neural networks

Recurrent neural networks

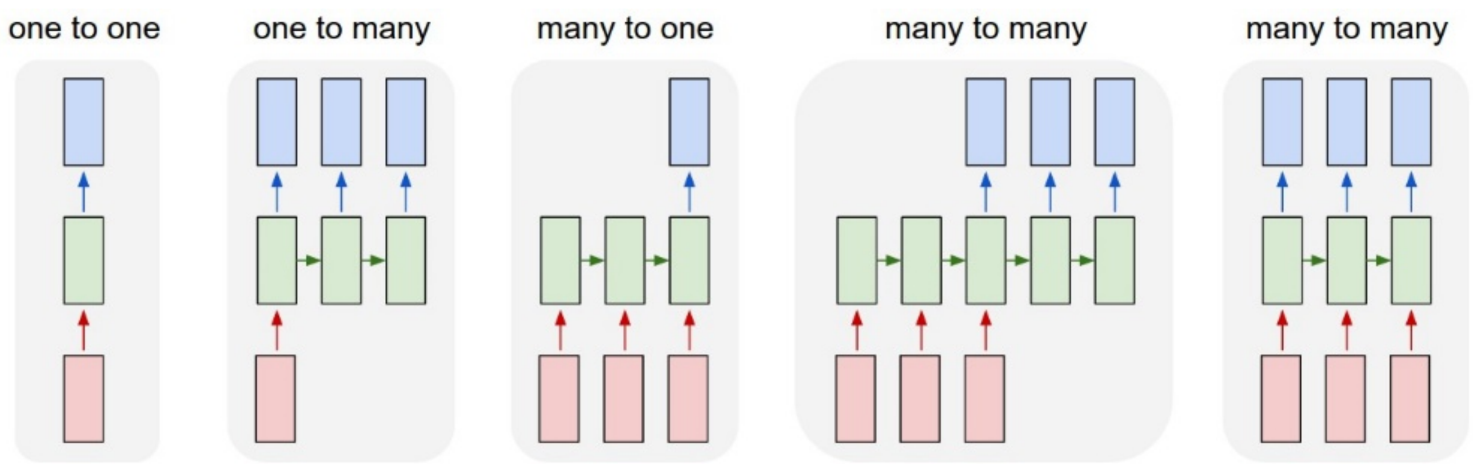
- Lots of information is sequential and requires a memory for successful processing
- Sequences as input, sequences as output



- Recurrent neural networks(RNNs) are called recurrent because they perform same task for every element of sequence, with output dependent on previous computations
- RNNs have memory that captures information about what has been computed so far
- RNNs can make use of information in arbitrarily long sequences – in practice they limited to looking back only few steps

Img Source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Topologies of Recurrent Neural Network



(1) (2) (3) (4) (5)

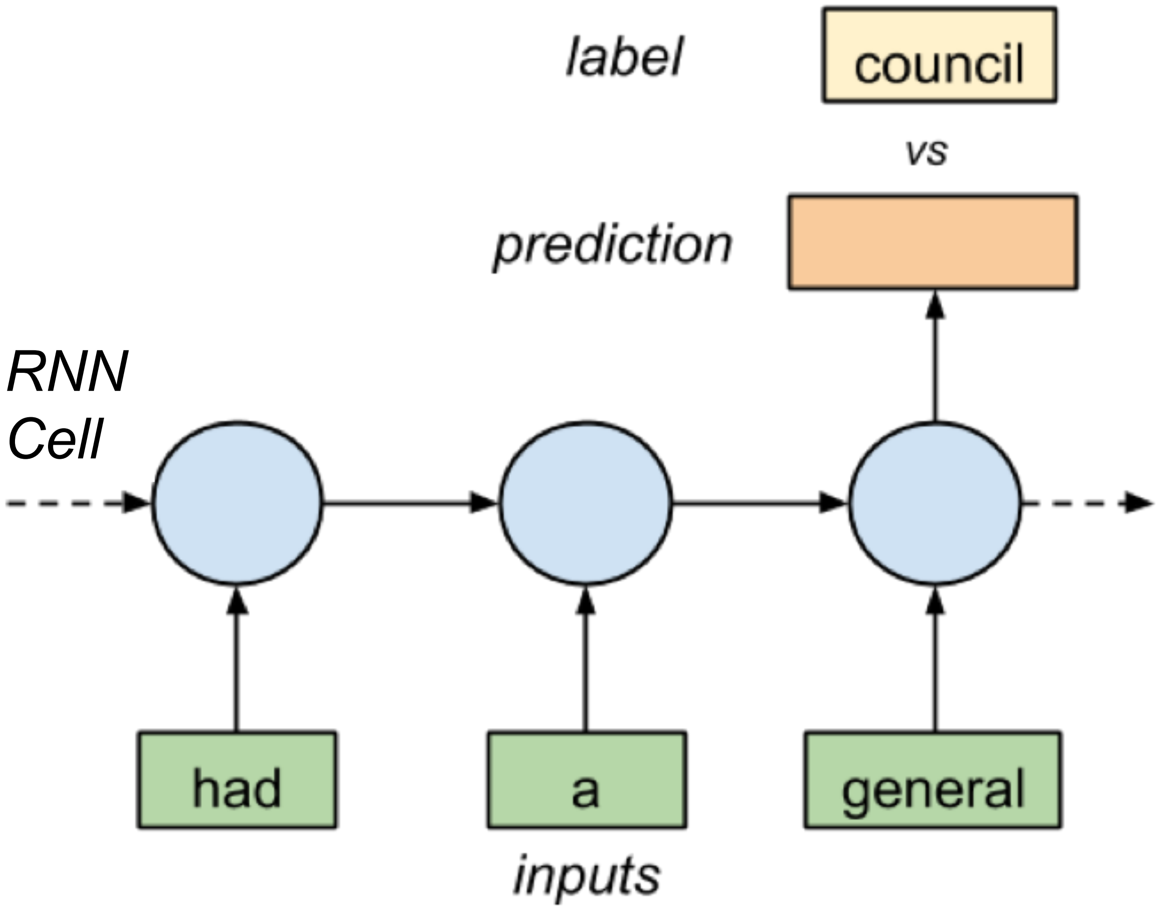
1) Common Neural Network (e.g. feed forward network)
2) Prediction of future states base on single observation
3) Sentiment classification
4) Machine translation
5) Simultaneous interpretation

Language Model

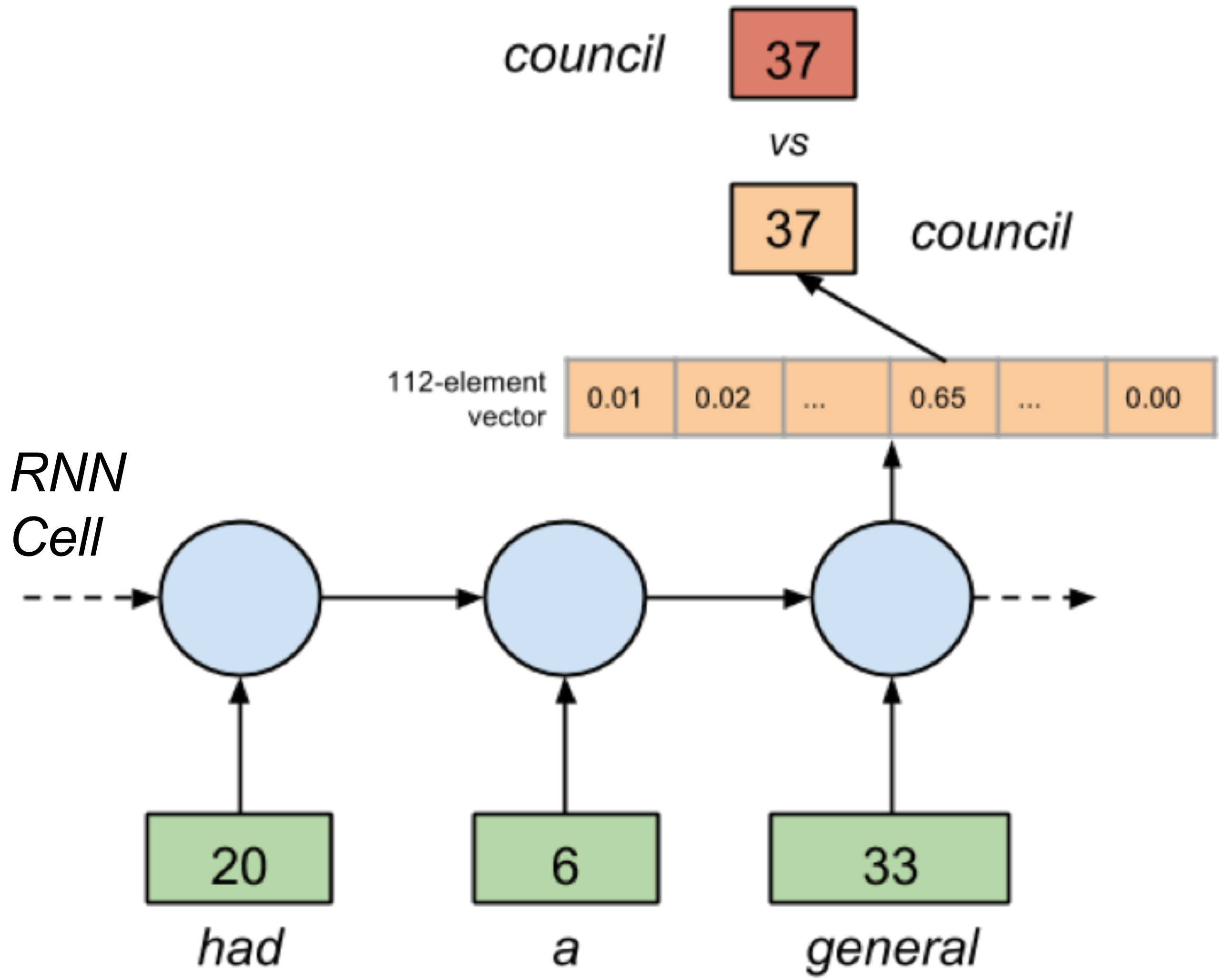
- Compute the probability of a sentence

- Useful in machine translation
  - Word ordering: p(the cat is small) > p(small the cat is)
  - Word choice: p(walking home after school) > p(walking house after school)

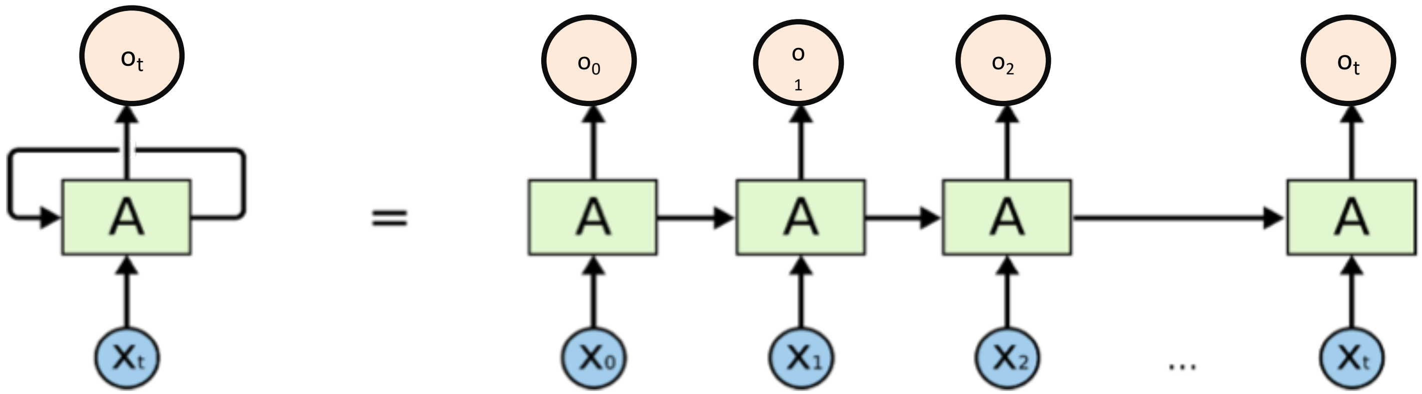
Recurrent Neural Network



Recurrent Neural Network



Recurrent Neural Network

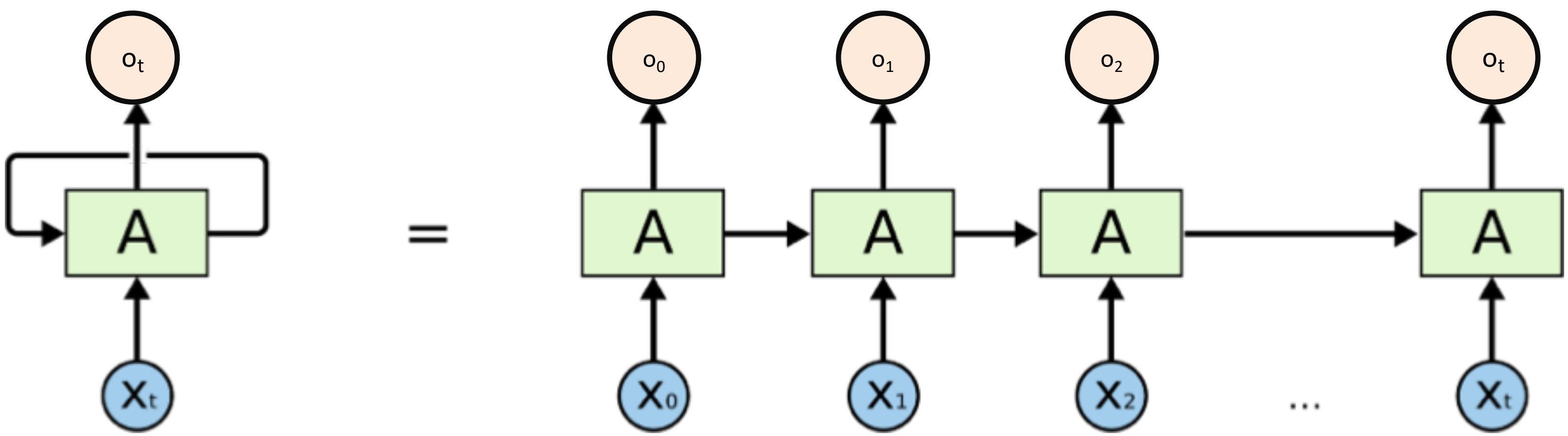


- Recurrent Neural Network have an internal state
- State is passed from input x_t to x_{t+1}

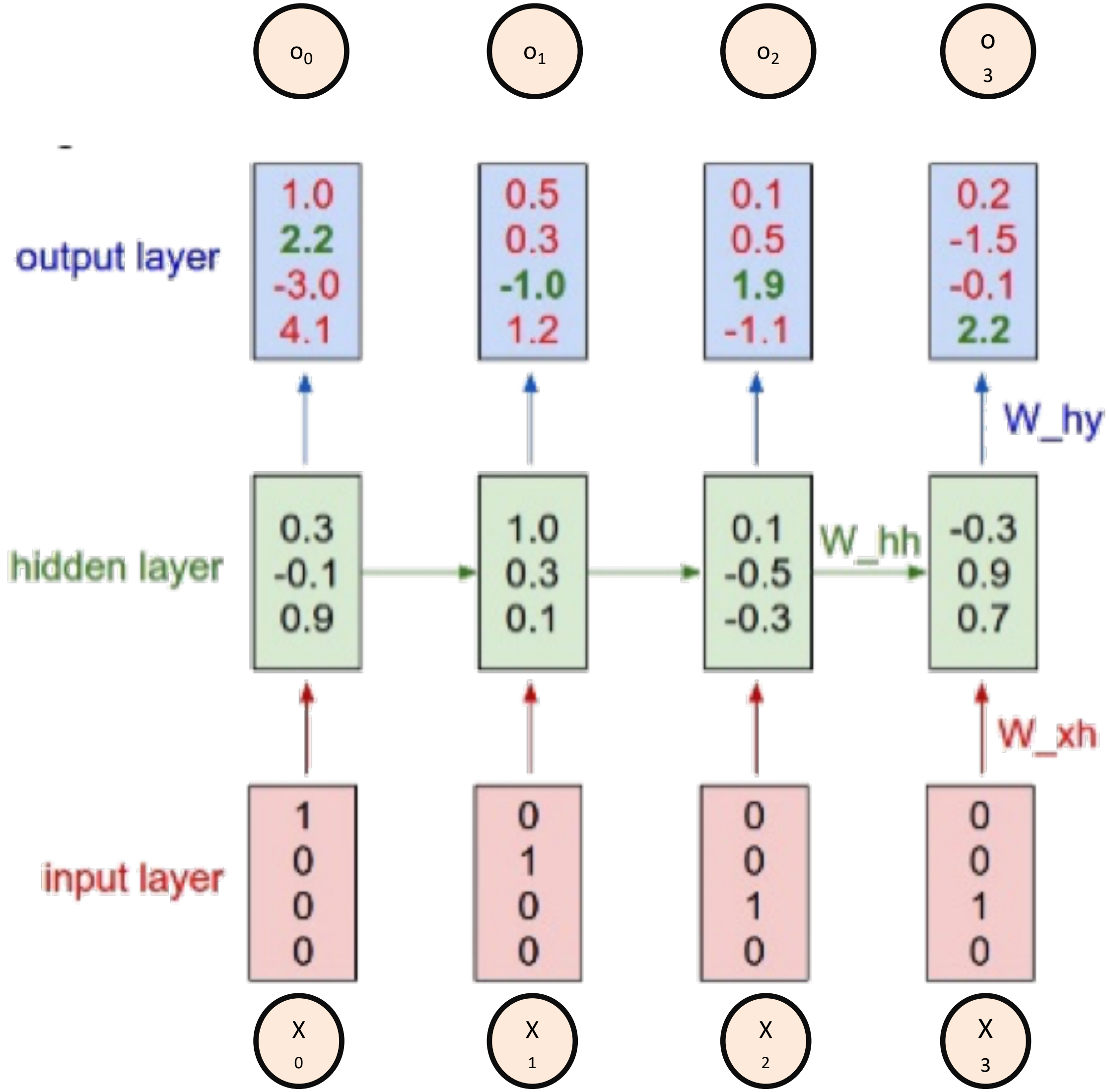
Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Language Models with RNN

- Let x_0, x_1, x_2… denote words (input)
- Let o_0, o_1, o_2… denote the probability of the sentence(output)
- Memory requirement scales nicely (linear with the number of word embeddings / number of character)

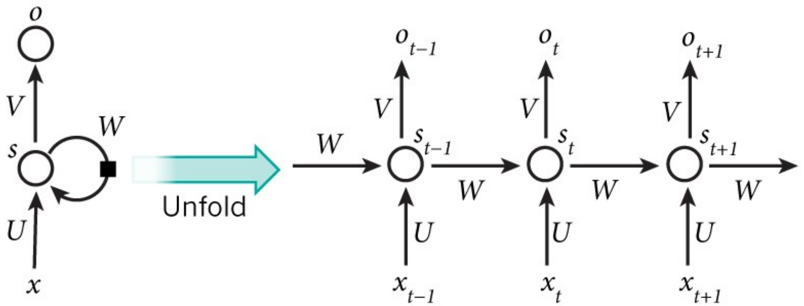


Recurrent Neural Network



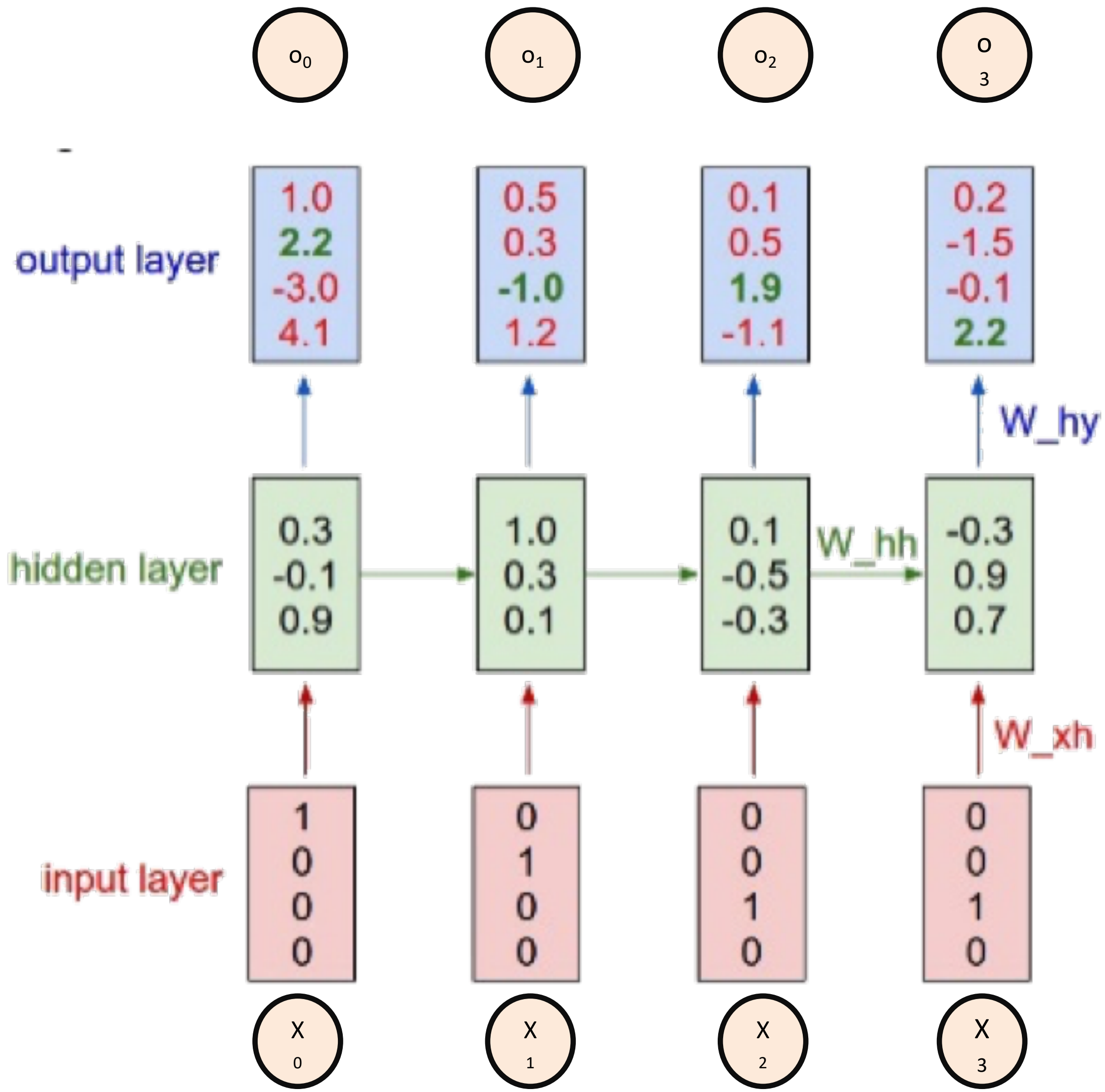
Recurrent neural networks

- RNN being unrolled (or unfolded) into full network
- Unrolling: write out network for complete sequence



- Image credits: Nature

RNN (Problem Revisited)



Img Source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

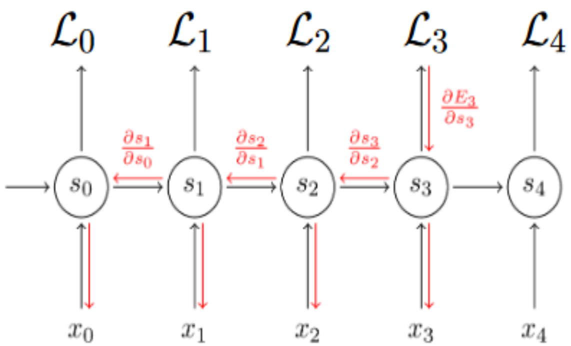
No Magic Involved (in Theory)

- You unroll your data in time
- You compute the gradients
- You use back propagation to train your network
- Karpathy presents a Python implementation for Char-RNN with 112 lines

- Training RNNs is hard:
  - Inputs from many time steps ago can modify output
  - Vanishing / Exploding Gradient Problem

- Vanishing gradients can be solved by Gated-RNNs like Long-Short-Term-Memory (LSTM) Models
  - LSTM became popular in NLP in 2015

Vanishing and exploding gradients



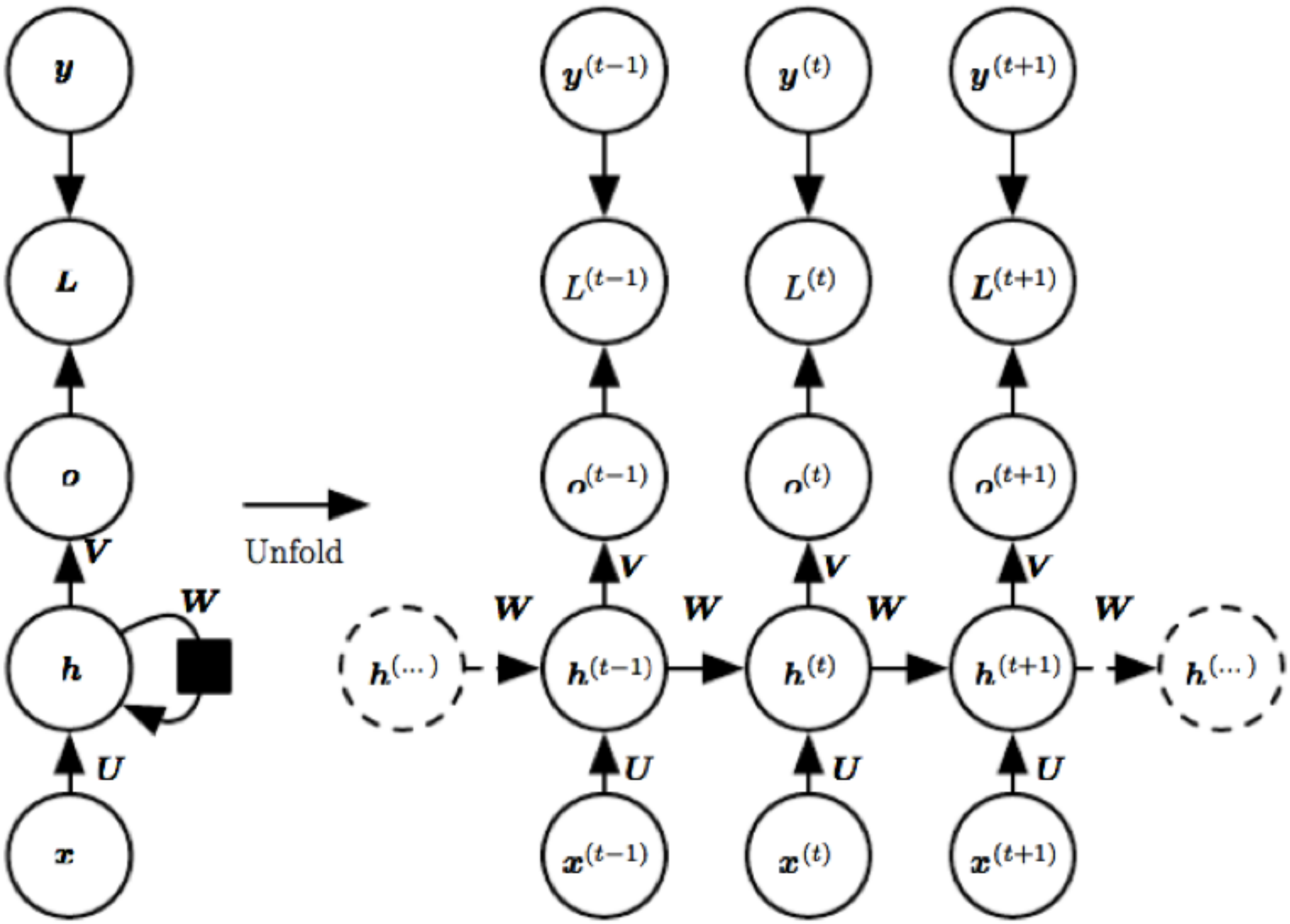
- For training RNNs, calculate gradients for U, V, W – ok for V but for W and U ...
- Gradients for W:

$$\frac{\partial \mathcal{L}_3}{\partial W} = \frac{\partial \mathcal{L}_3}{\partial o_3}\frac{\partial o_3}{\partial s_3}\frac{\partial s_3}{\partial W} = \sum_{k=0}^{3}\frac{\partial \mathcal{L}_3}{\partial o_3}\frac{\partial o_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$

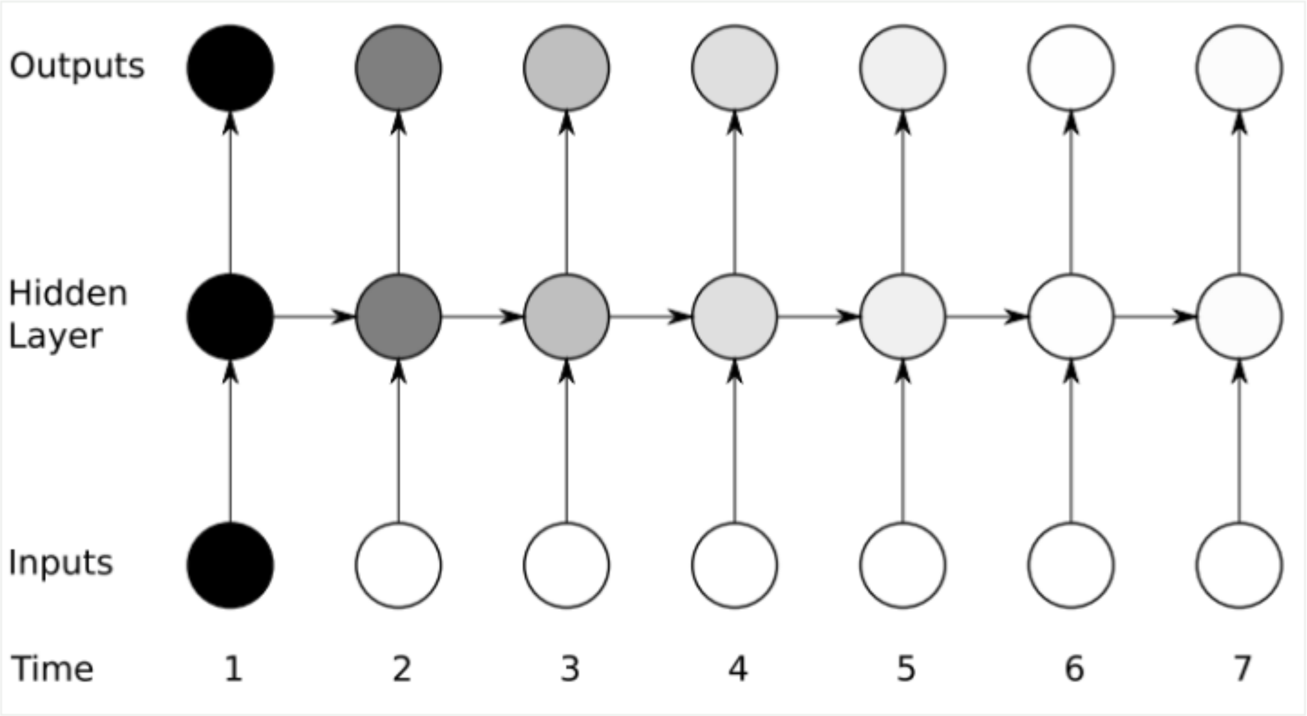
- More generally: $\frac{\partial \mathcal{L}}{\partial s_t} = \frac{\partial \mathcal{L}}{\partial s_m} \cdot \underbrace{\frac{\partial s_m}{\partial s_{m-1}}}_{<1} \cdot \underbrace{\frac{\partial s_{m-1}}{\partial s_{m-2}}}_{<1} \cdot \cdots \cdot \underbrace{\frac{\partial s_{t+1}}{\partial s_t}}_{<1} \Rightarrow \ll 1$
- Gradient contributions from far away steps become zero: state at those steps doesn't contribute to what you are learning

L_i – Loss, U, V, W – Parameters, S_i - states

Vanishing and exploding gradients



Vanishing and exploding gradients



Heatmap

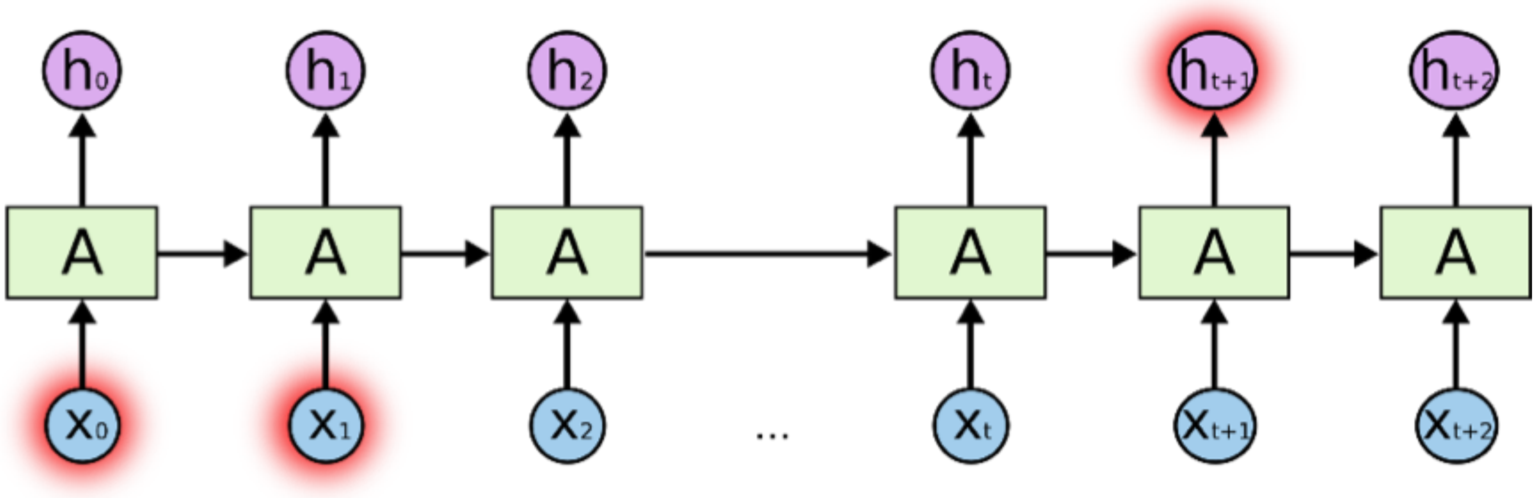
Long Short Term Memory [Hochreiter and Schmidhuber, 1997]

LSTMs designed to combat vanishing gradients through gating mechanism

How LSTM calculates hidden state s_t

$$i = \sigma(x_t U^i + s_{t-1} W^i)$$
$$f = \sigma(x_t U^f + s_{t-1} W^f)$$
$$o = \sigma(x_t U^o + s_{t-1} W^o)$$
$$g = \tanh(x_t U^g + s_{t-1} W^g)$$
$$c_t = c_{t-1} \circ f + g \circ i$$
$$s_t = \tanh(c_t) \circ o$$

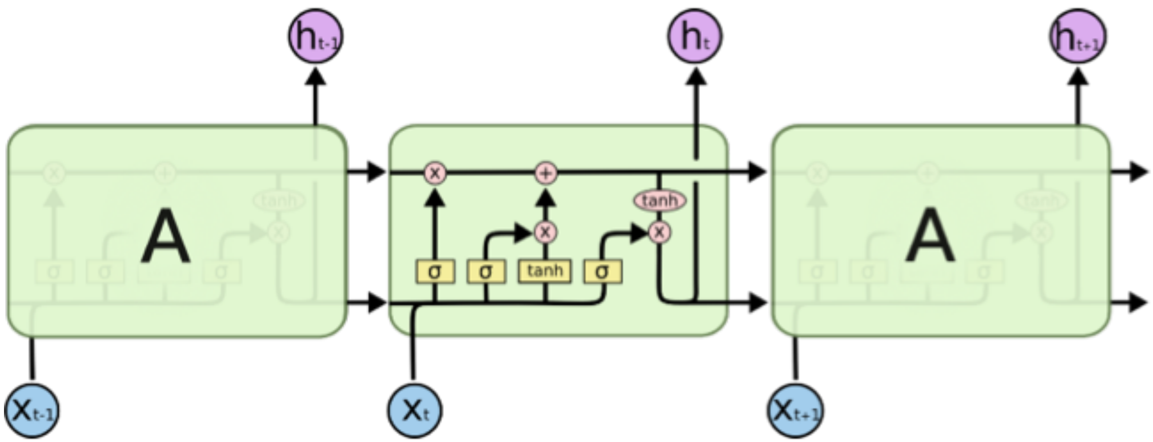
Long-Short-Term Memory (LSTM)



- Long-term dependencies:
  I grew up in France and lived there until I was 18. Therefore I speak fluent ???
- Presented (vanilla) RNN is unable to learn long term dependencies
  - Issue: More recent input data has higher influence on the output
- Long-Short-Term Memory (LSTM) models solves this problem

Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

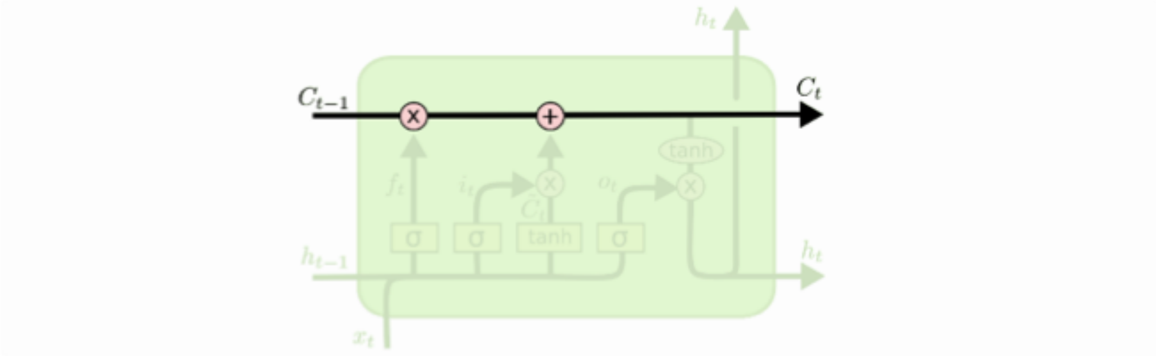
LSTM Model



- The LSTM model implements a *forget-gate* and an *add-gate*
- The models learns when to forget something and when to update internal storage

Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

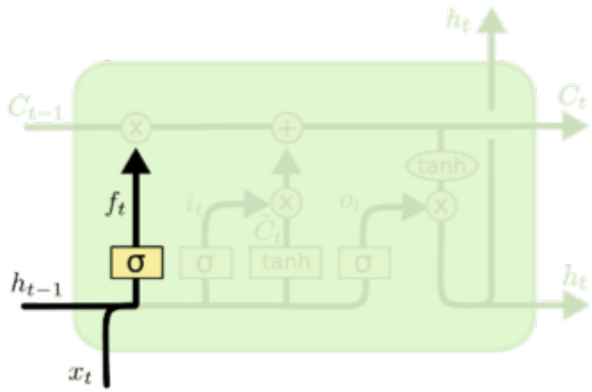
LSTM Model



- Core: Cell-state C (a vector of certain size)
- The model has the ability to remove or add information using Gates

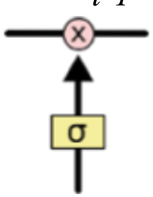
Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Forget-Gate



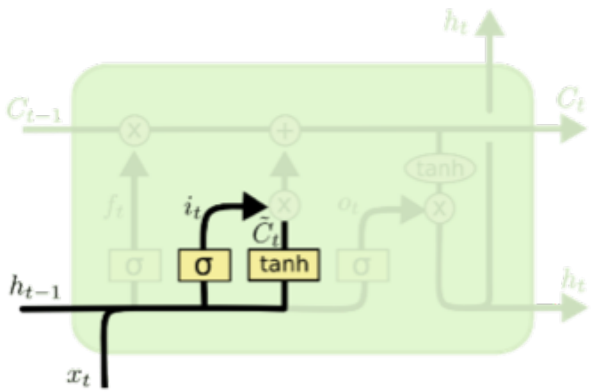
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

- Sigmoid function σ output a value between 0 and 1
- The output is point-wise multiplied with the cell state C_{t-1}
- Interpretation:
  - 0: *Let nothing through*
  - 1: *Let everything through*



- Example: When we see a new subject, forget gender of old subject

Set-Gate

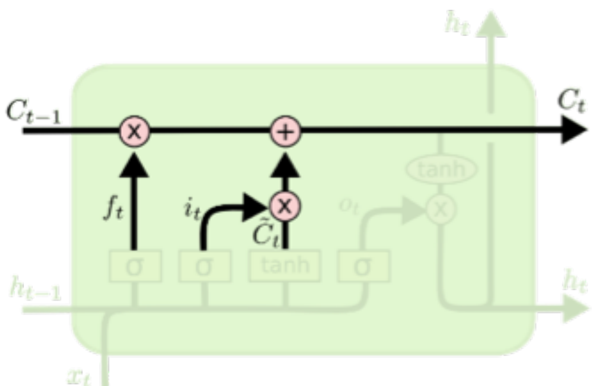


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

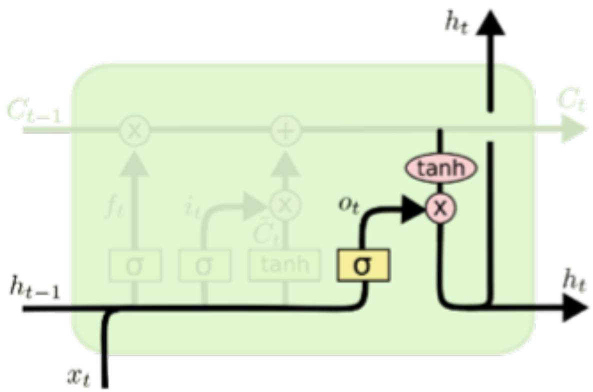
- Compute i_t which cells we want to update and to which degree (σ: 0 … 1)
- Compute the new cell value using the *tanh* function

Update Internal Cell State



$$C_t = \underbrace{f_t * C_{t-1}}_{\text{Forget state cells}} + \underbrace{i_t * \tilde{C}_t}_{\text{Update state cells}}$$

Compute Output h_t



$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

- We use the updated cell state C_t to compute the output
- We might not need the complete cell state as output
  - Compute o_t, defining how relevant each cell is for the output
  - Pointwise multiply o_t with $tanh(C_t)$

- Cell state C_t and output h_t is passed to the next time step

Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Conclusion

- Deep learning approaches – Powerful mechanisms for introducing non-linearity in learning
- Learning using backpropagation
- Embeddings for word representations
- Sequence Labelling using RNNs
- LSTMs, GRUs are special kind of RNNs
- CNNs for text and Image recognition.

References

- **Deep Learning for NLP - Nils Reimers.** https://github.com/UKPLab/deeplearning4nlp-tutorial/tree/master/2017-07_Seminar
- CS231n: Convolutional Neural Networks for Visual Recognition. Andrej Karpathy http://cs231n.github.io/convolutional-networks/
- http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- **Neural Networks for Information Retrieval**. SIGIR 2017 Tutorial http://nn4ir.com/
- **CSE 446 - Machine Learning - Spring 2015,** University of Washington. Pedro Domingos. https://courses.cs.washington.edu/courses/cse446/15sp/

- Thanks