DATA MINING 2
(Deep) Neural Networks

Riccardo Guidotti

a.a. 2020/2021

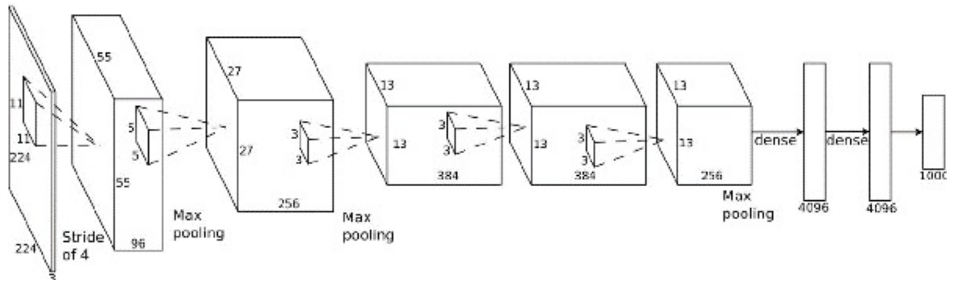
Slides edited from a set of slides titled "Introduction to Machine Learning and Neural Networks" by Davide Bacciu

UNIVERSITÀ DI PISA

Why Now?

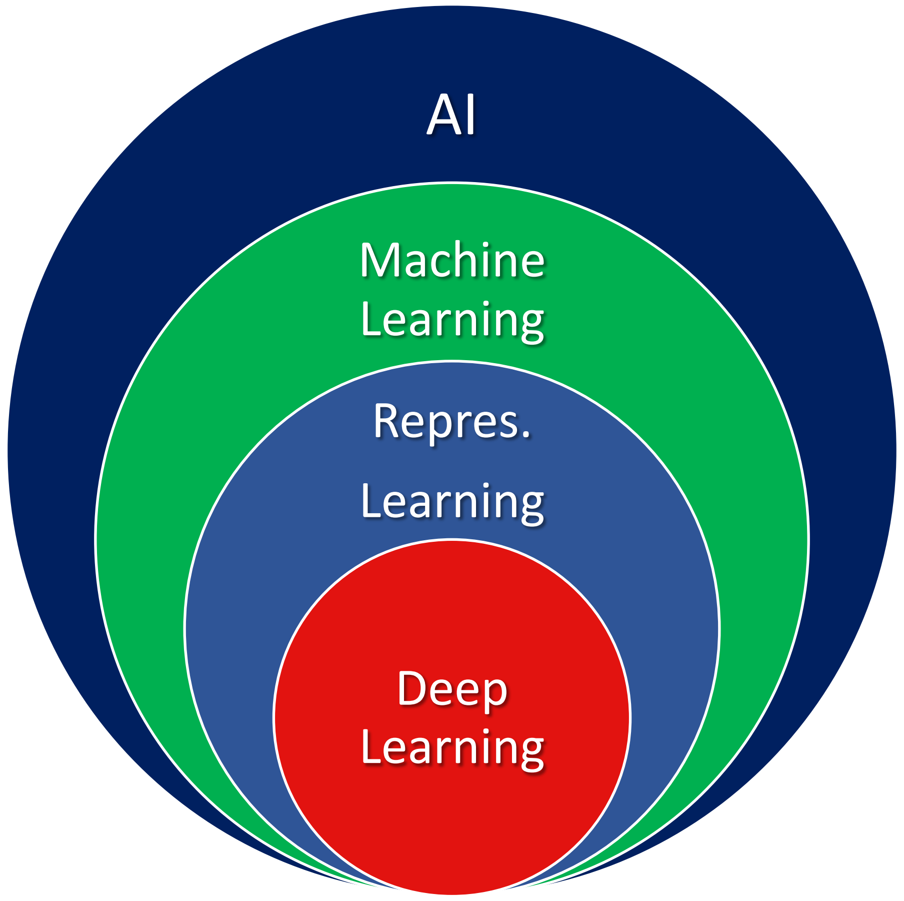
(Big) Data

GPU

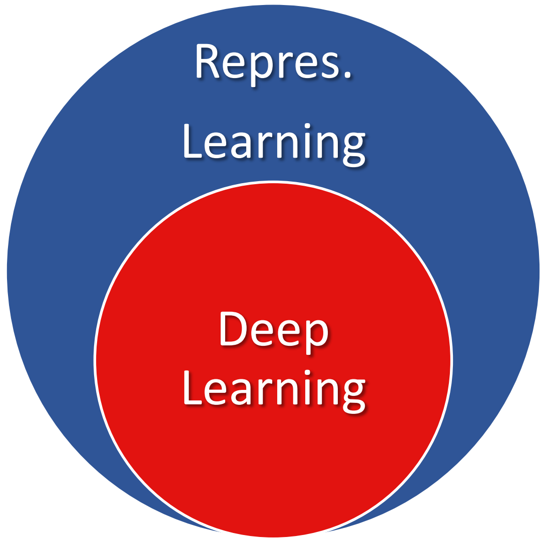


Theory

A quick look on Deep Learning



Deep learning

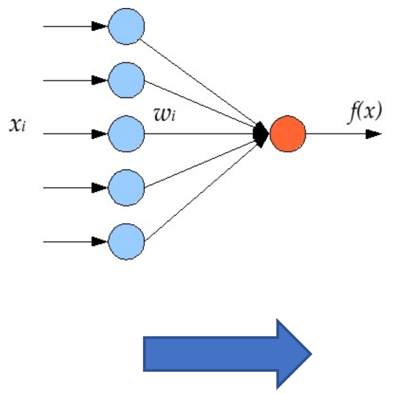


Representation learning methods that

- allow a machine to be fed with raw data and
- to automatically discover the representations needed for detection or classification.

Raw representation

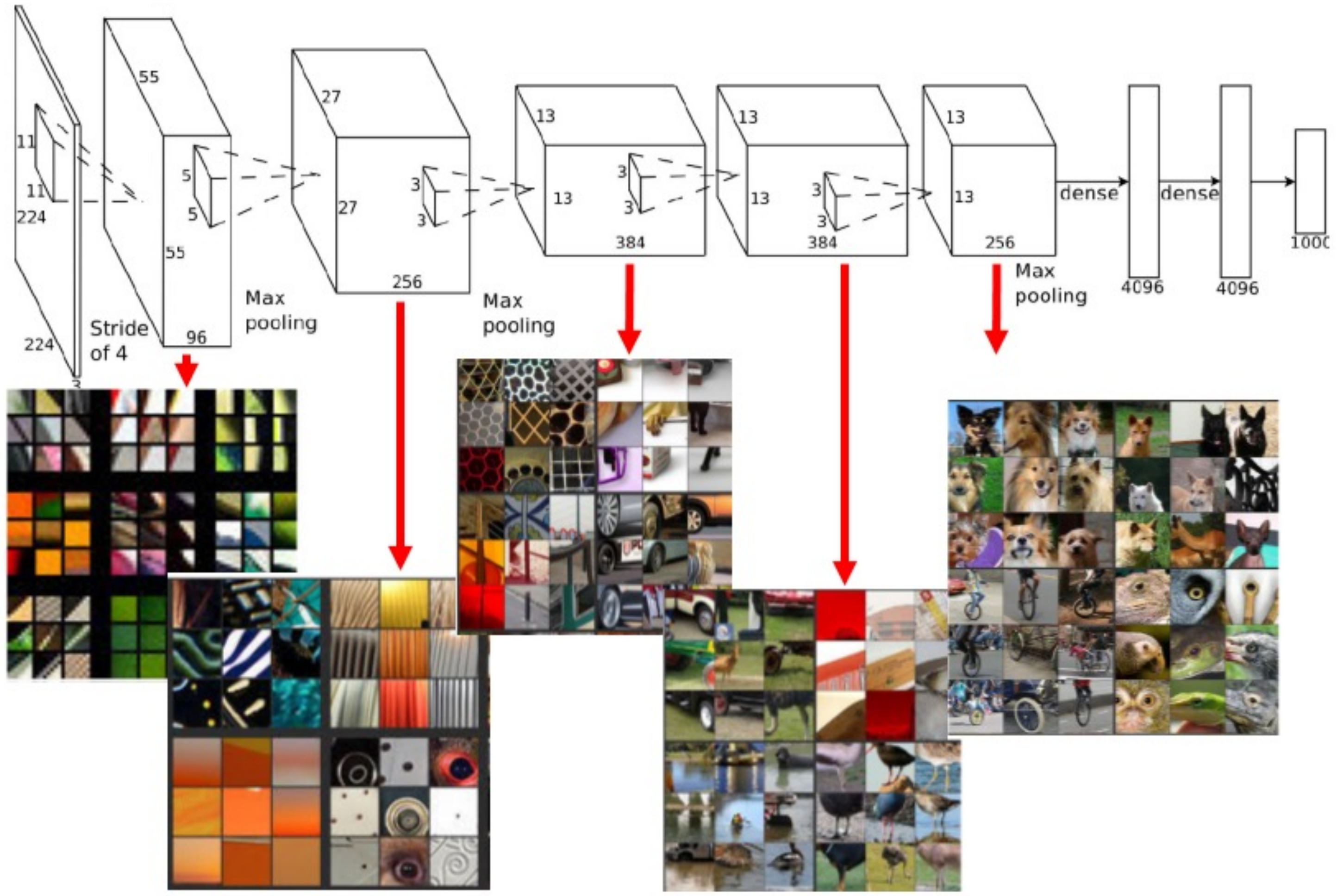
Age	35
Weight	65
Income	23 k€
Children	2
Likes sport	0.3
Likes reading	0.6
Education	high
…	…



Higher-level representation

Young parent	0.9
Fit sportsman	0.1
High-educated reader	0.8
Rich obese	0.0
…	…

Multiple Levels Of Abstraction



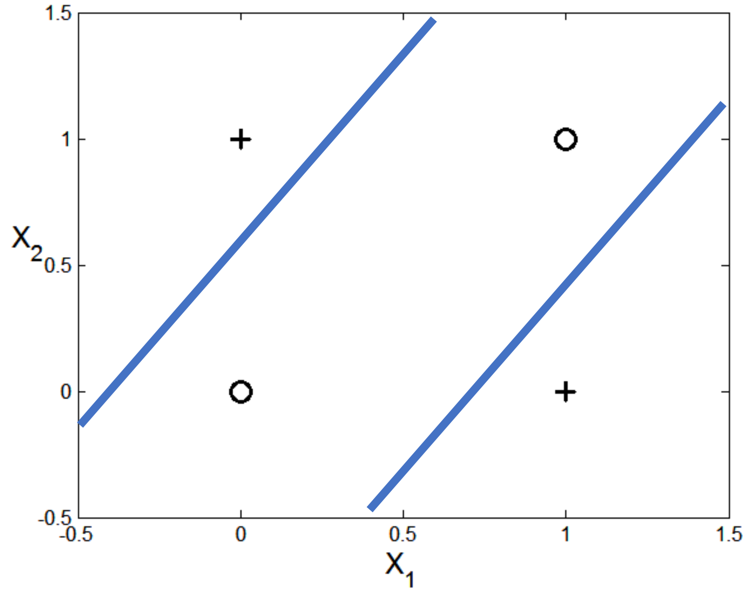
Nonlinearly Separable Data

- Since *f(w,x)* is a linear combination of input variables, decision boundary is linear.
- For nonlinearly separable problems, the perceptron fails because no linear hyperplane can separate the data perfectly.
- An example of nonlinearly separable data is the XOR function.

XOR Data

x_1	x_2	y
0	0	-1
1	0	1
0	1	1
1	1	-1

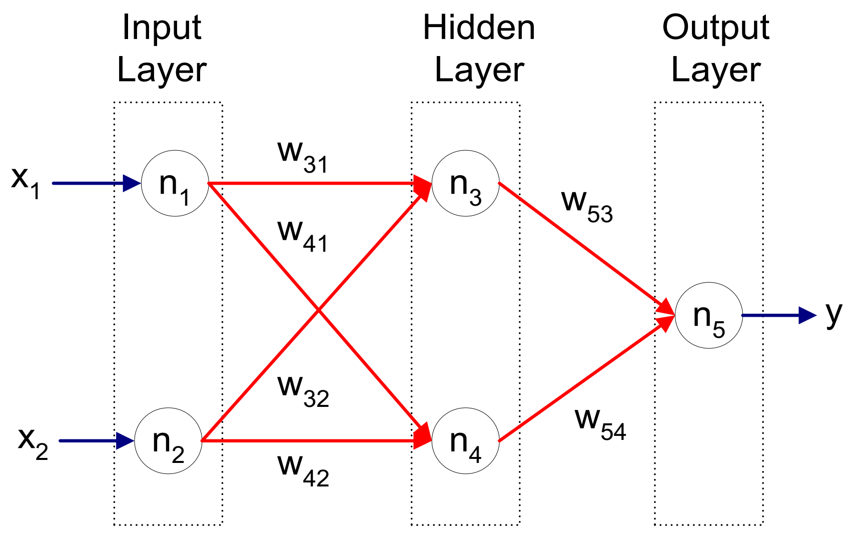
$$y = x_1 \oplus x_2$$

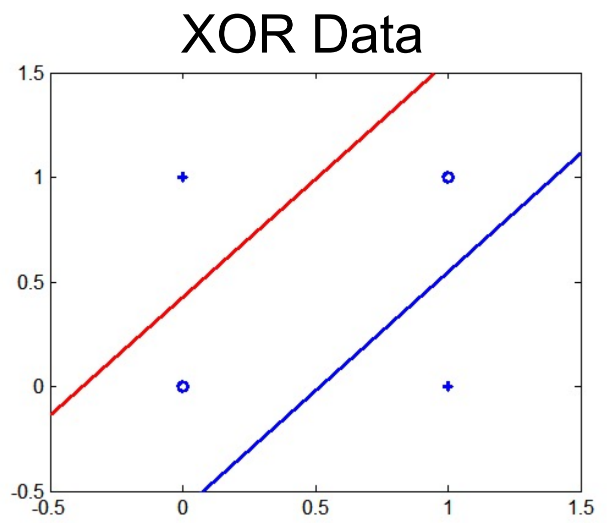


Multilayer Neural Network

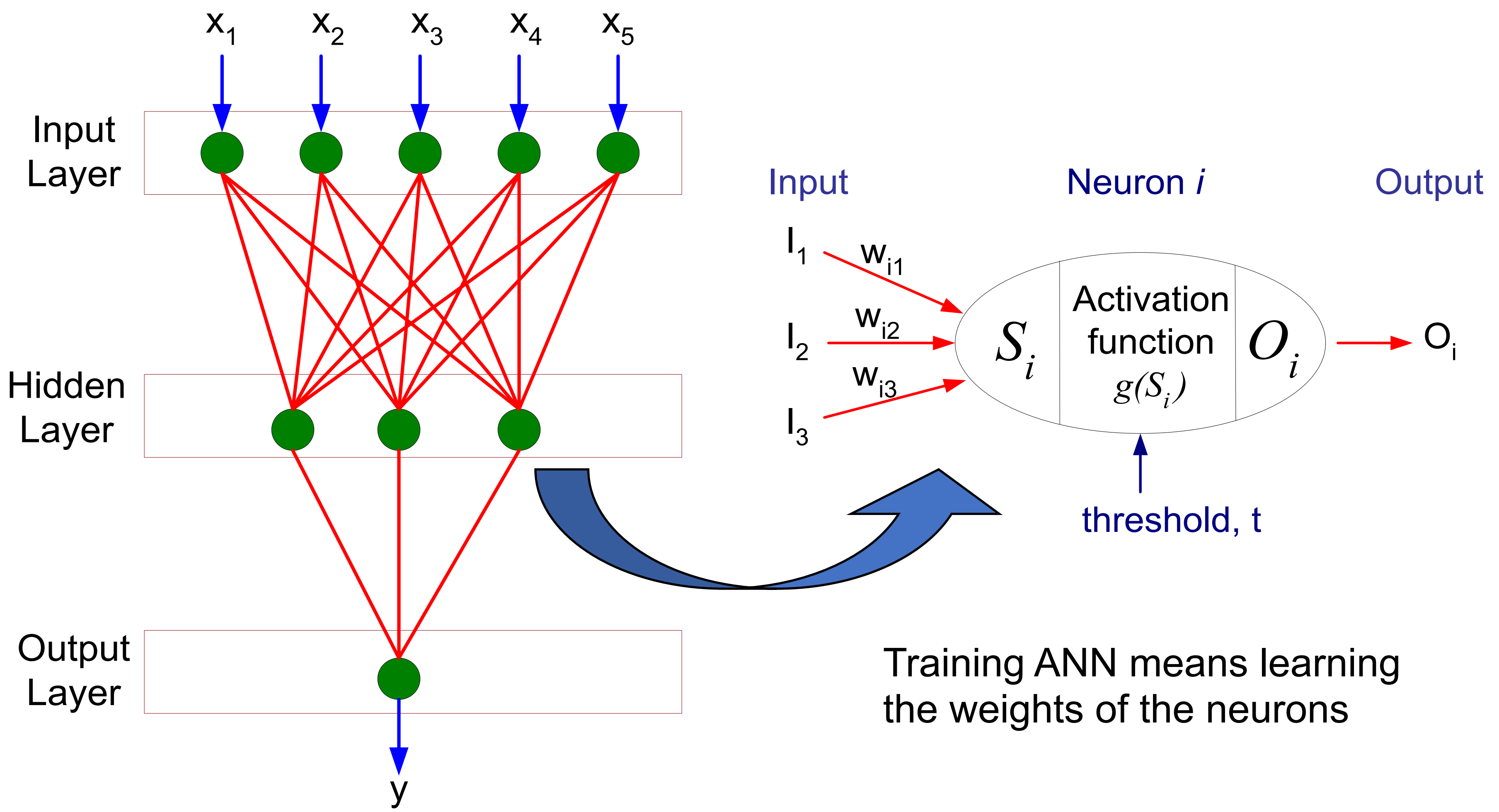
- **Hidden Layers**: intermediary layers between input and output layers.
- More general **activation functions** (sigmoid, linear, hyperbolic tangent, etc.).

- Multi-layer neural network can solve any type of classification task involving nonlinear decision surfaces.
- Perceptron is single layer.
- We can think to each hidden node as a perceptron that tries to construct one hyperplane, while the output node combines the results to return the decision boundary.





General Structure of ANN



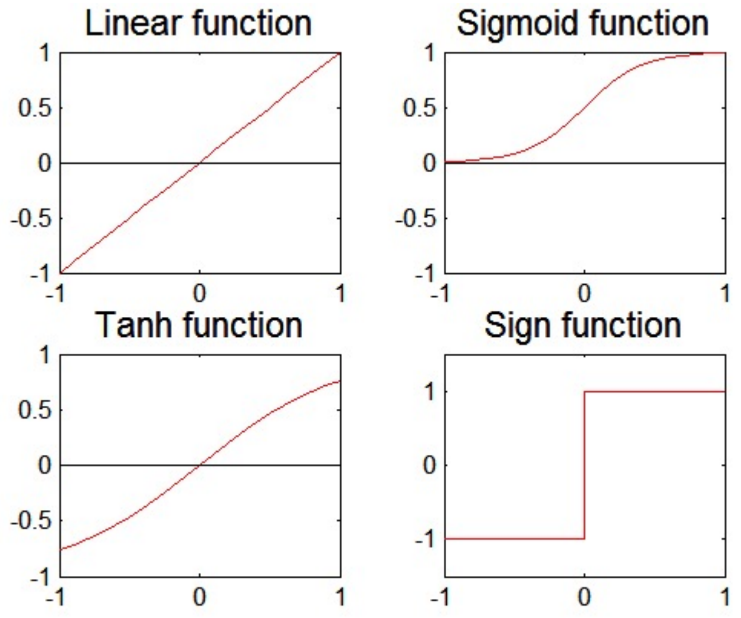
Training ANN means learning the weights of the neurons

Artificial Neural Networks (ANN)

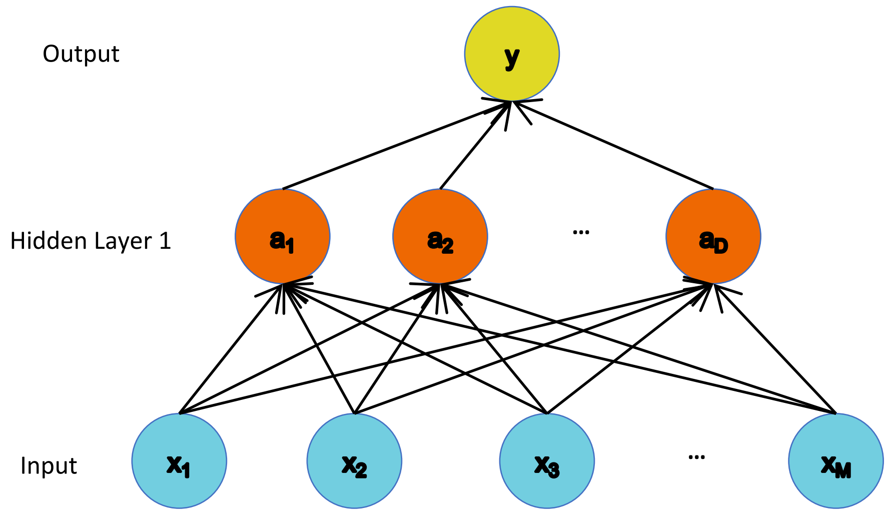
- Various types of neural network topology
  - single-layered network (perceptron) versus multi-layered network
  - Feed-forward versus recurrent network

- Various types of activation functions (f)

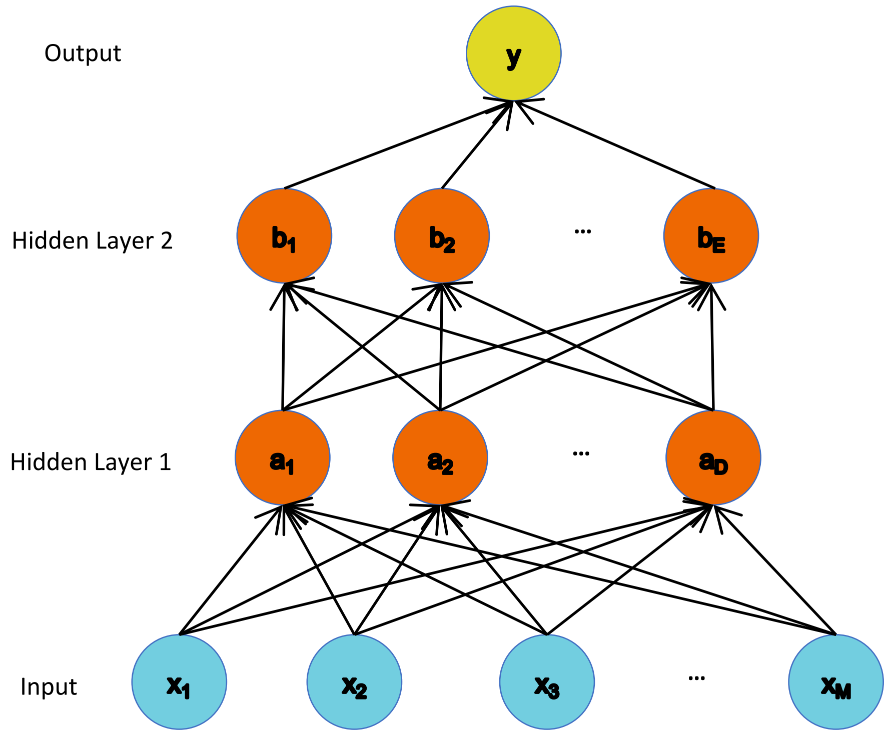
$$Y = f\left(\sum_i w_i X_i\right)$$



Deep Neural Networks

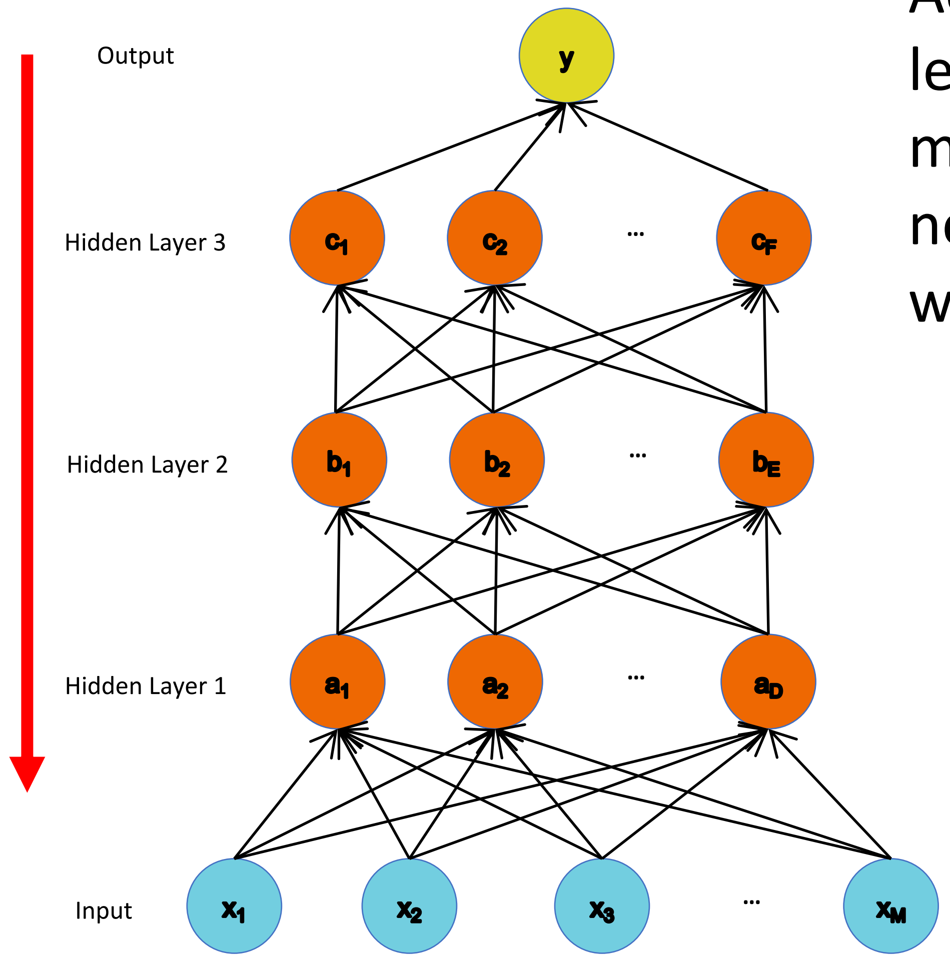


Deep Neural Networks



Deep Neural Networks

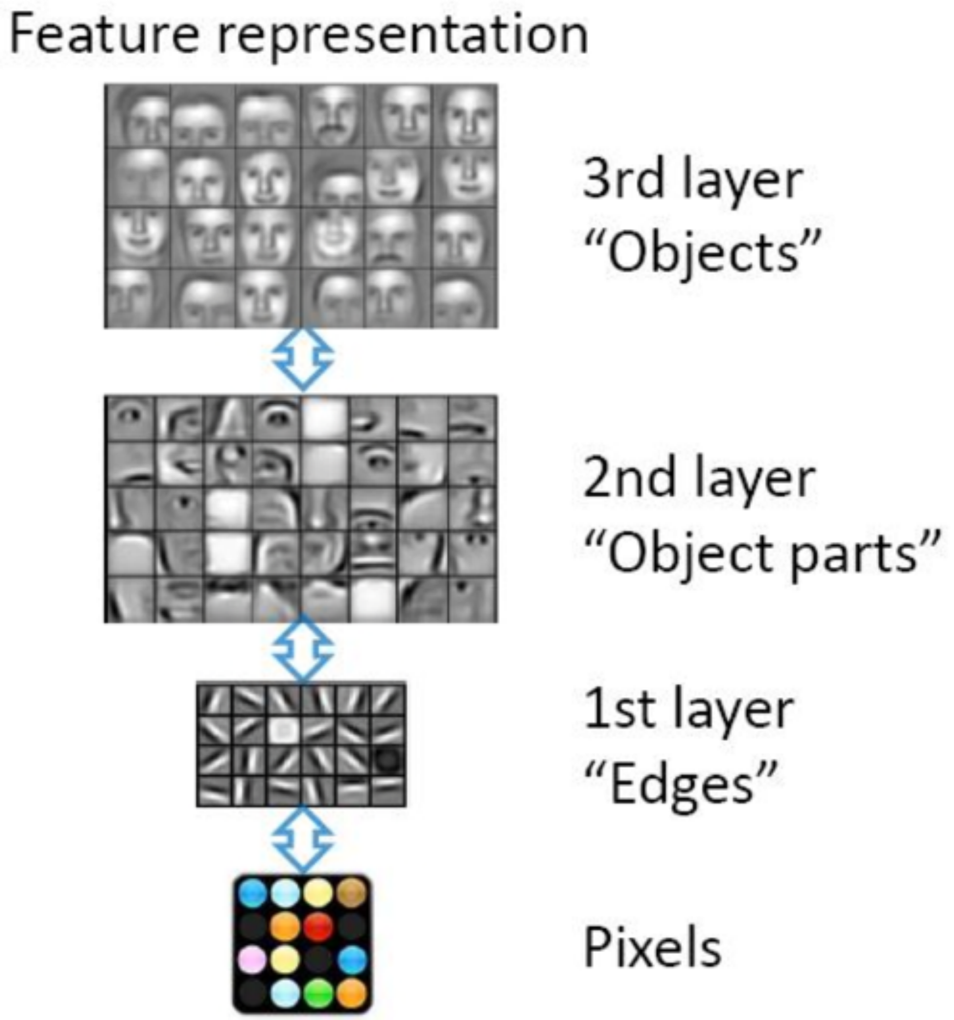
Backpropagation through many layers has numerical problems that makes learning not-straightforward (Gradient Vanish/Esplosion)



Actually deep learning is way more than having neural networks with a lot of layers

Representation Learning

- We don’t know the “right” levels of abstraction of information that is good for the machine
- So let the model figure it out!

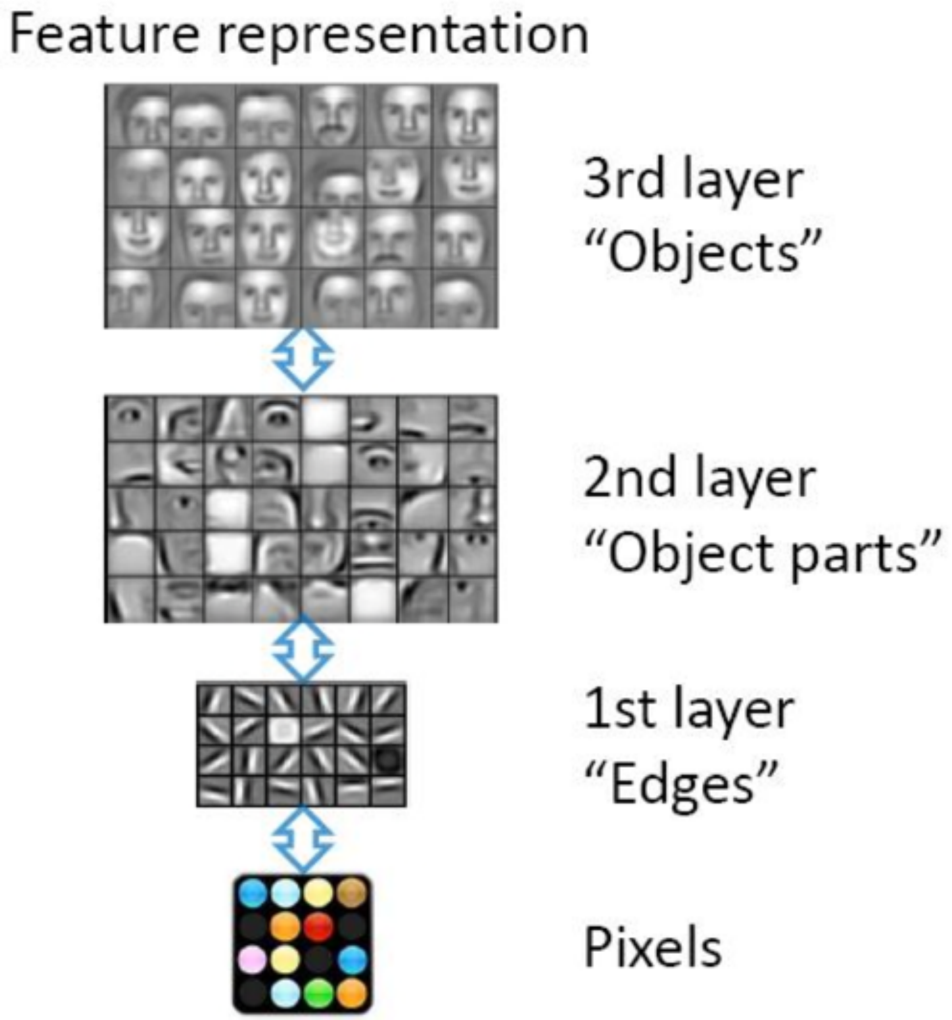


Example from Honglak Lee (NIPS 2010)

Representation Learning

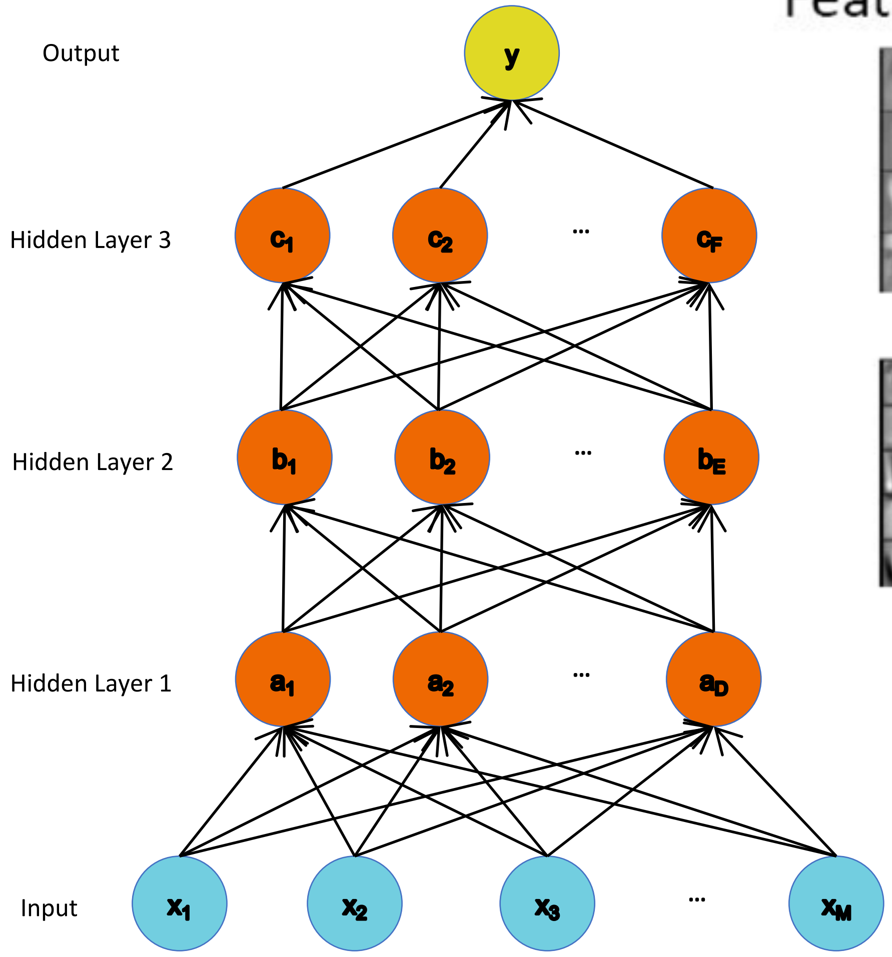
Face Recognition:

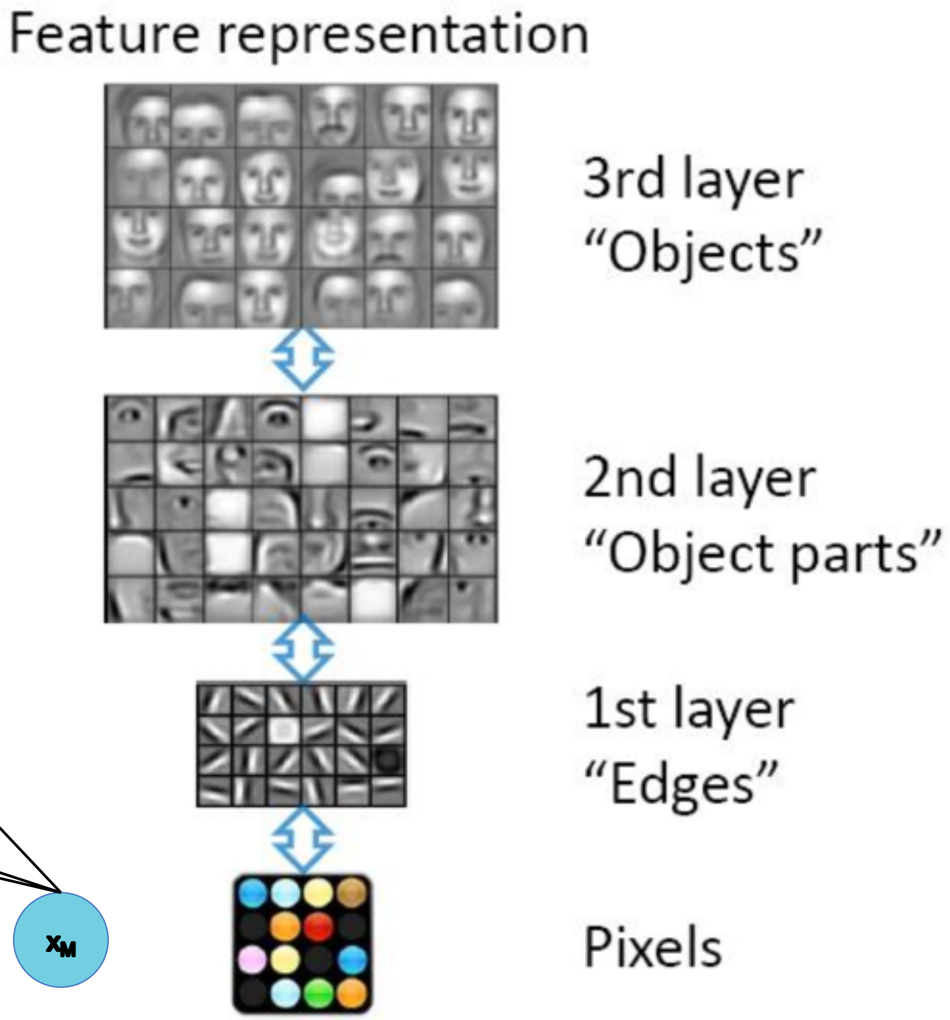
- Deep Network can build up increasingly higher levels of abstraction
- Lines, parts, regions



Example from Honglak Lee (NIPS 2010)

Representation Learning

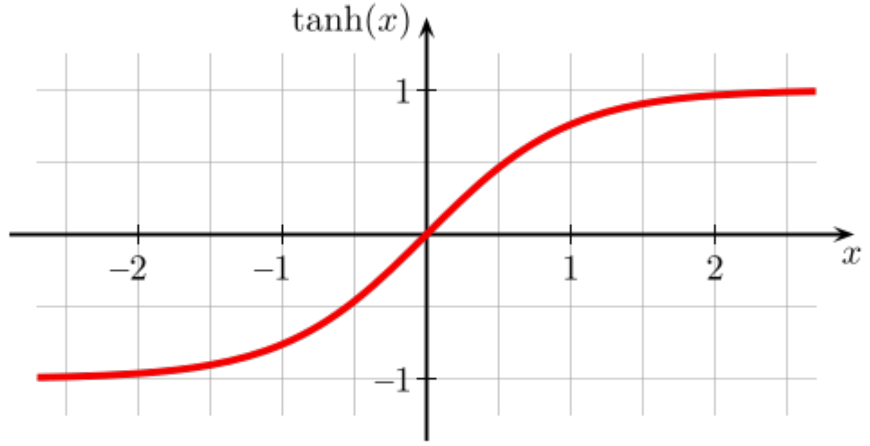




Example from Honglak Lee (NIPS 2010)

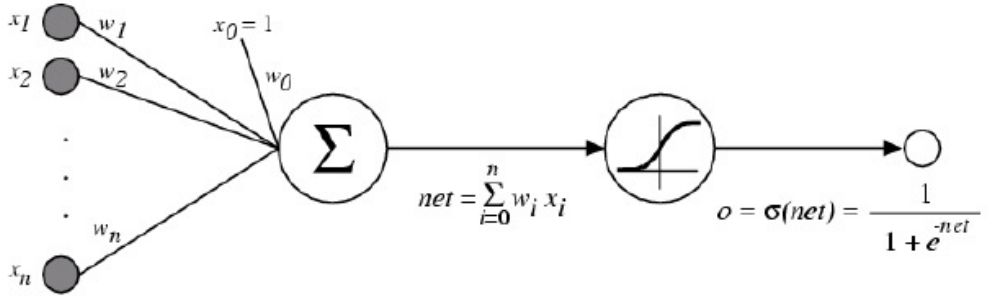
Activation Functions

- A new change: modifying the nonlinearity
  - The logistic is not widely used in modern ANNs

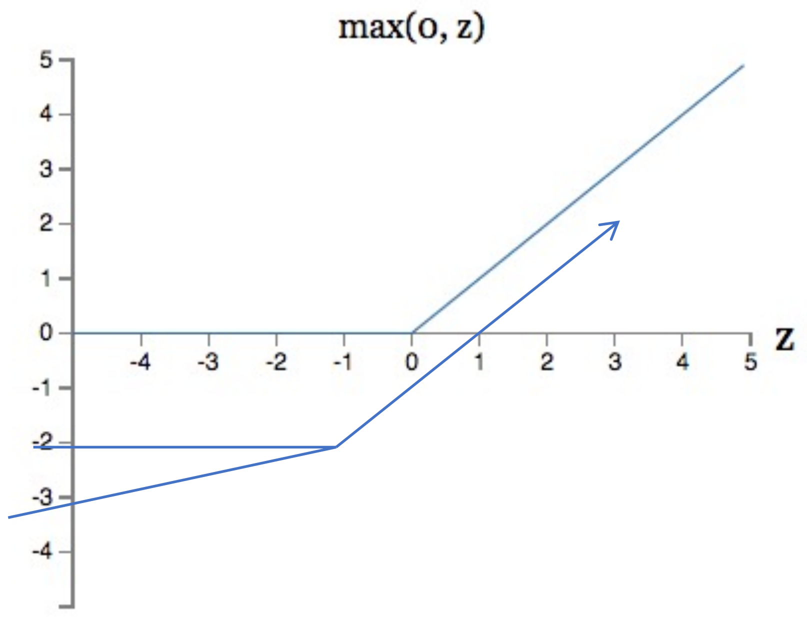


Alternative 1:
tanh

Like logistic function but shifted to range [-1, +1]


x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1+e^{-net}}$

Activation Functions



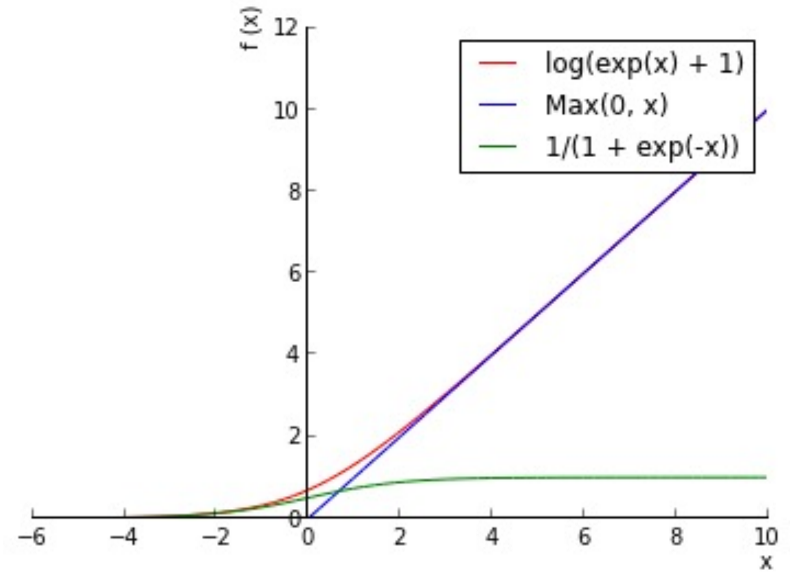
Alternative 2: rectified linear unit

Linear with a cutoff at zero

(Implementation: clip the gradient when you pass zero)

$$\max(0, w \cdot x + b).$$

Activation Functions



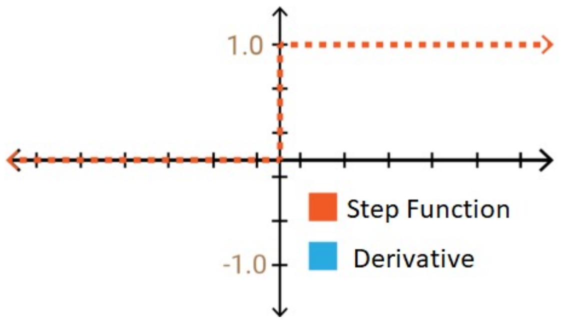
Alternative 3: soft exponential linear unit

Soft version: log(exp(x)+1)

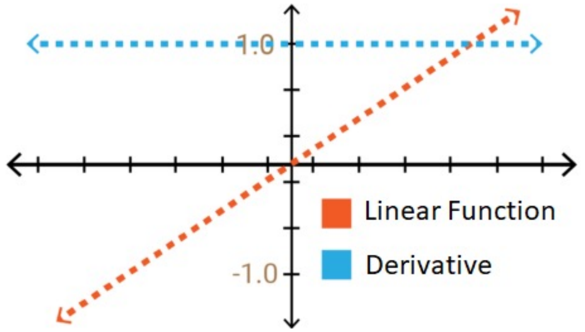
Doesn’t saturate (at one end)
Sparsifies outputs
Helps with vanishing gradient

Slide from William Cohen

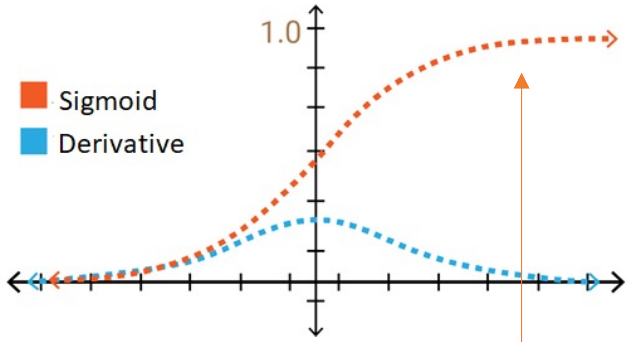
Activation Functions Summary



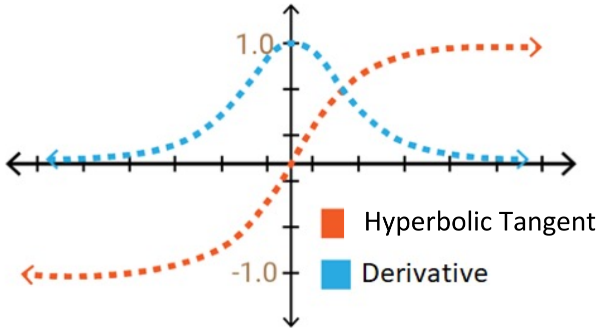
$$f(x) = \begin{cases} 0 \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$$



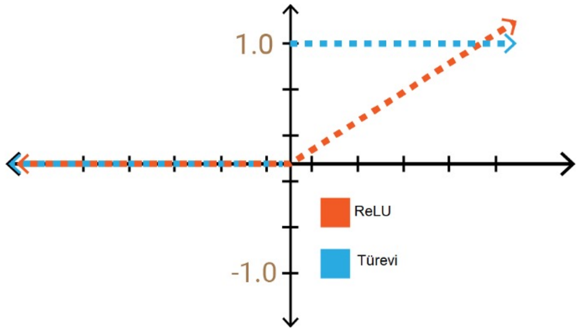
$$f(x) = x$$



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



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



$$f(x) = \begin{cases} 0 \; (or \; \epsilon) \; for \; x < 0 \\ x \qquad\quad for \; x \geq 0 \end{cases}$$

$$f(x_j) = \frac{e^{x_j}}{\sum_k e^{x_k}}$$

Softmax Function

Learning Multi-layer Neural Network

- Can we apply perceptron learning to each node, including hidden nodes?
- Perceptron computes error $e = y - f(w,x)$ and updates weights accordingly
- Problem: how to determine the true value of *y* for hidden nodes?
- Approximate error in hidden nodes by error in the output nodes
- Problems:
    - Not clear how adjustment in the hidden nodes affect overall error
    - No guarantee of convergence to optimal solution

Gradient Descent for Multilayer NN

- Error function to minimize: $E = \frac{1}{2}\sum_{i=1}^{N}\left(y_i - f(\sum_j w_j x_{ij})\right)^2$

Quadratic function from which we can find a global minimum solution

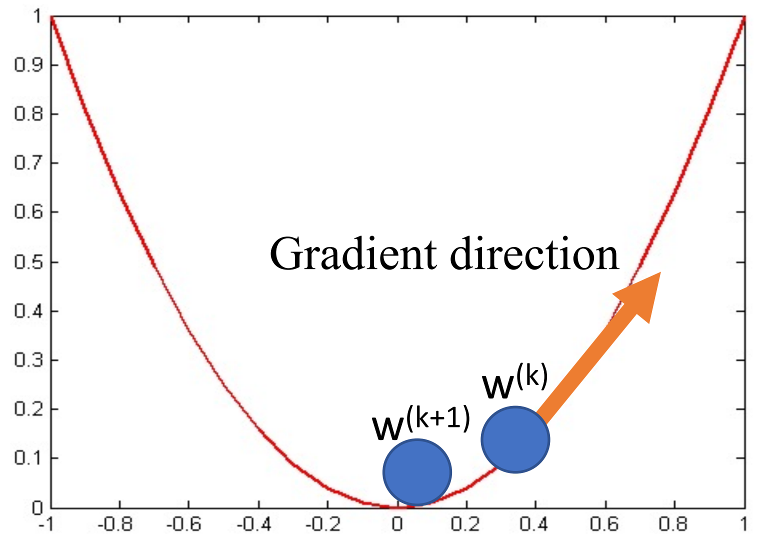
- Weight update: $w_j^{(k+1)} = w_j^{(k)} - \lambda \frac{\partial E}{\partial w_j}$
- Activation function *f* must be differentiable
- For sigmoid function: $w_j^{(k+1)} = w_j^{(k)} + \lambda \sum_i (y_i - o_i) o_i (1 - o_i) x_{ij}$
- Stochastic Gradient Descent (update the weight immediately)

Gradient Descent for Multilayer NN

- Weights are updated in the opposite direction of the gradient of the loss function.

$$w_j^{(k+1)} = w_j^{(k)} - \lambda \frac{\partial E}{\partial w_j}$$

- Gradient direction is the direction of uphill of the error function.
- By taking the negative we are going downhill.
- Hopefully to a minimum of the error.



Gradient Descent for Multilayer NN

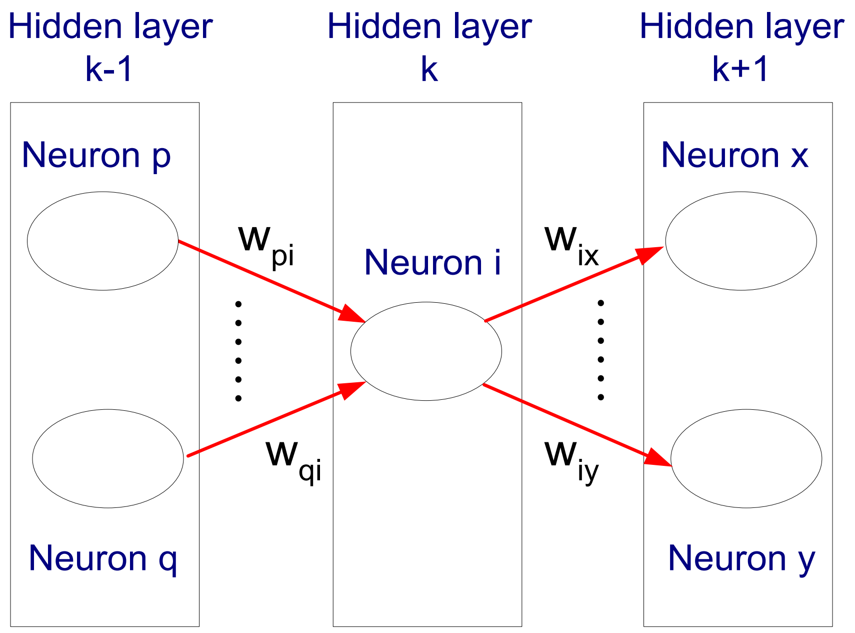
- For output neurons, weight update formula is the same as before (gradient descent for perceptron)

- For hidden neurons:

$$w_{pi}^{(k+1)} = w_{pi}^{(k)} + \lambda o_i(1-o_i)\sum_{j\in\Phi_i}\delta_j w_{ij} x_{pi}$$

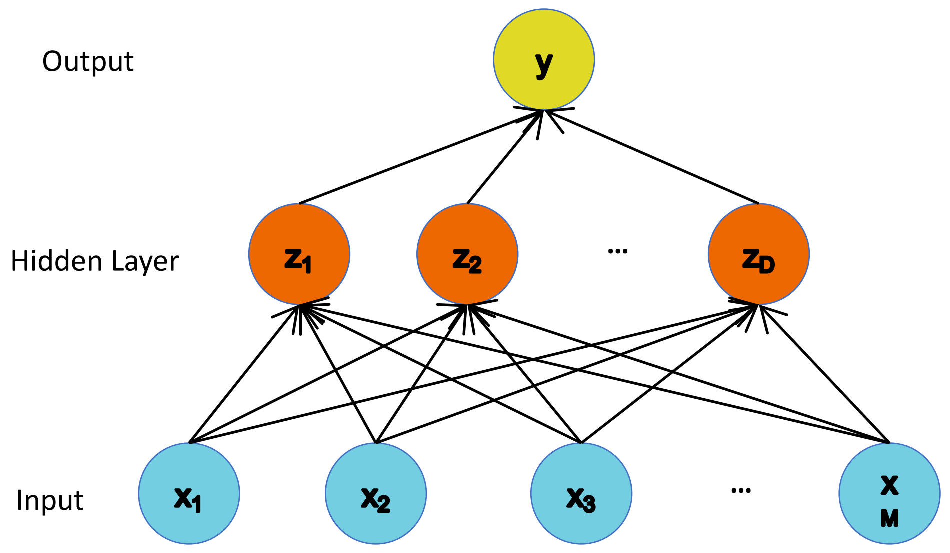
$$\text{Output neurons}: \delta_j = o_j(1-o_j)(t_j - o_j)$$

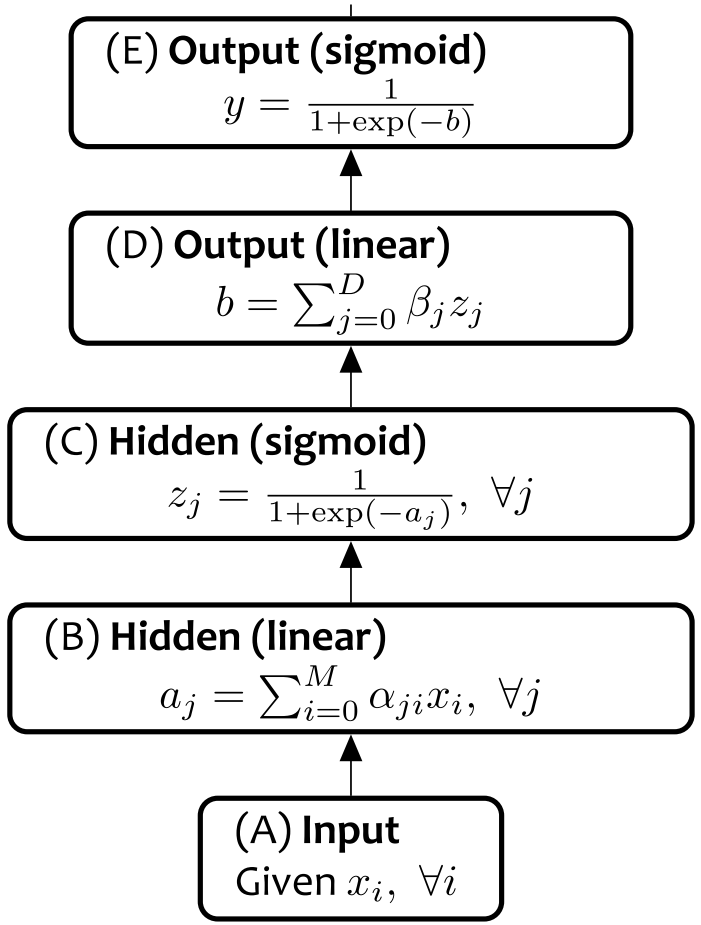
$$\text{Hidden neurons}: \delta_j = o_j(1-o_j)\sum_{k\in\Phi_j}\delta_k w_{jk}$$



o: output of the network
t: target value (ground truth)

Training Multilayer NN





(E) **Output (sigmoid)**
$$y = \frac{1}{1+\exp(-b)}$$

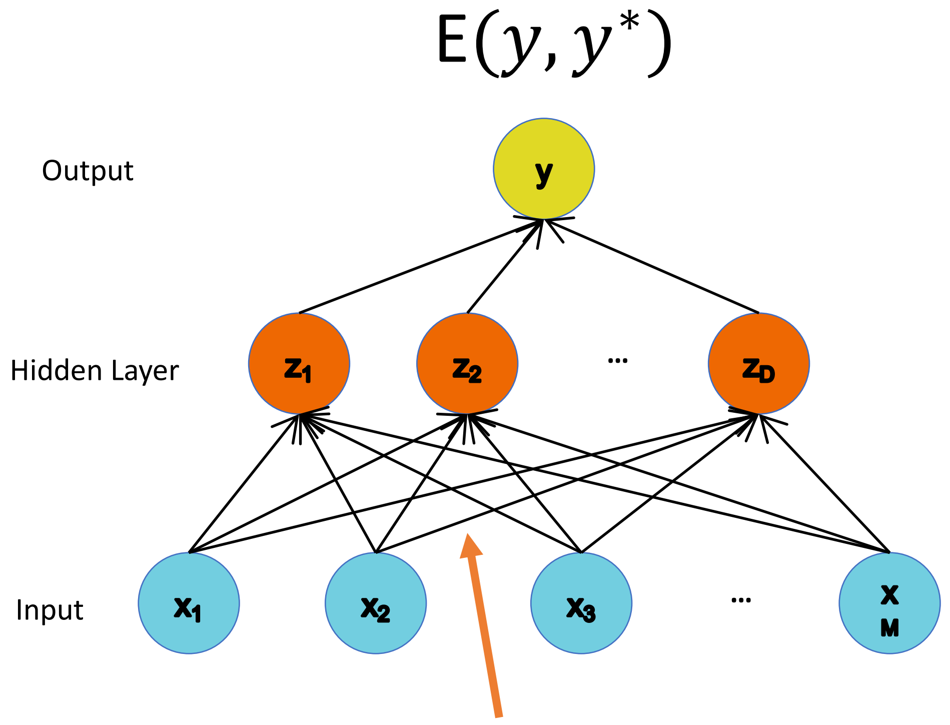
(D) **Output (linear)**
$$b = \sum_{j=0}^{D} \beta_j z_j$$

(C) **Hidden (sigmoid)**
$$z_j = \frac{1}{1+\exp(-a_j)}, \ \forall j$$

(B) **Hidden (linear)**
$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \ \forall j$$

(A) **Input**
Given $x_i, \ \forall i$

Training Multilayer NN



How do we update these weights given the loss is available only at the output unit?

(F) Loss
E $= \frac{1}{2}(y - y^*)^2$

(E) Output (sigmoid)
$y = \frac{1}{1+\exp(-b)}$

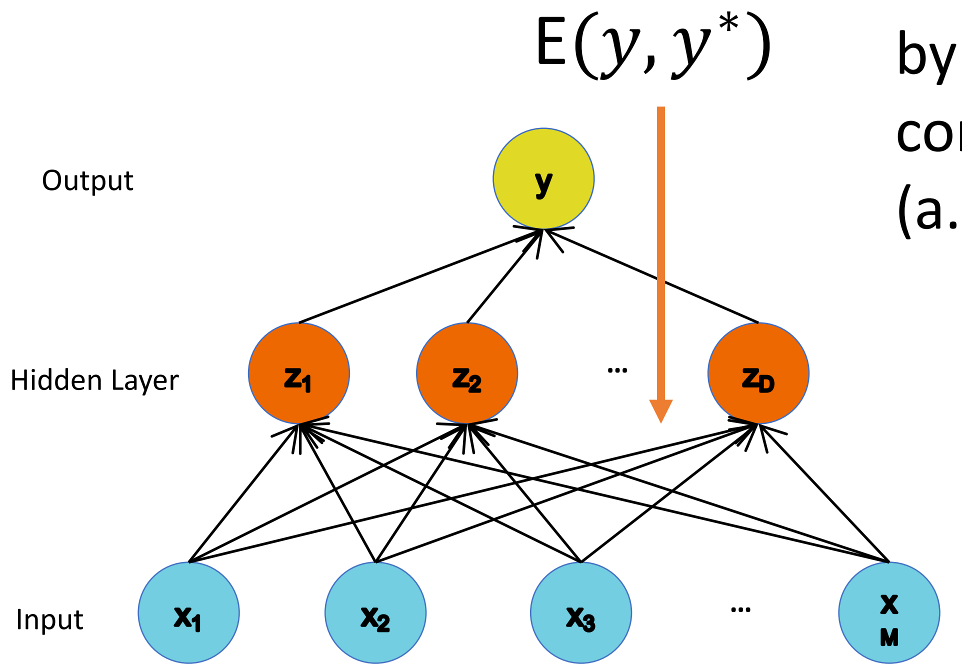
(D) Output (linear)
$b = \sum_{j=0}^{D} \beta_j z_j$

(C) Hidden (sigmoid)
$z_j = \frac{1}{1+\exp(-a_j)}, \ \forall j$

(B) Hidden (linear)
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \ \forall j$

(A) Input
Given $x_i, \ \forall i$

Error Backpropagation



Error is computed at the output and propagated back to the input by chain rule to compute the contribution of each weight (a.k.a. derivative) to the loss

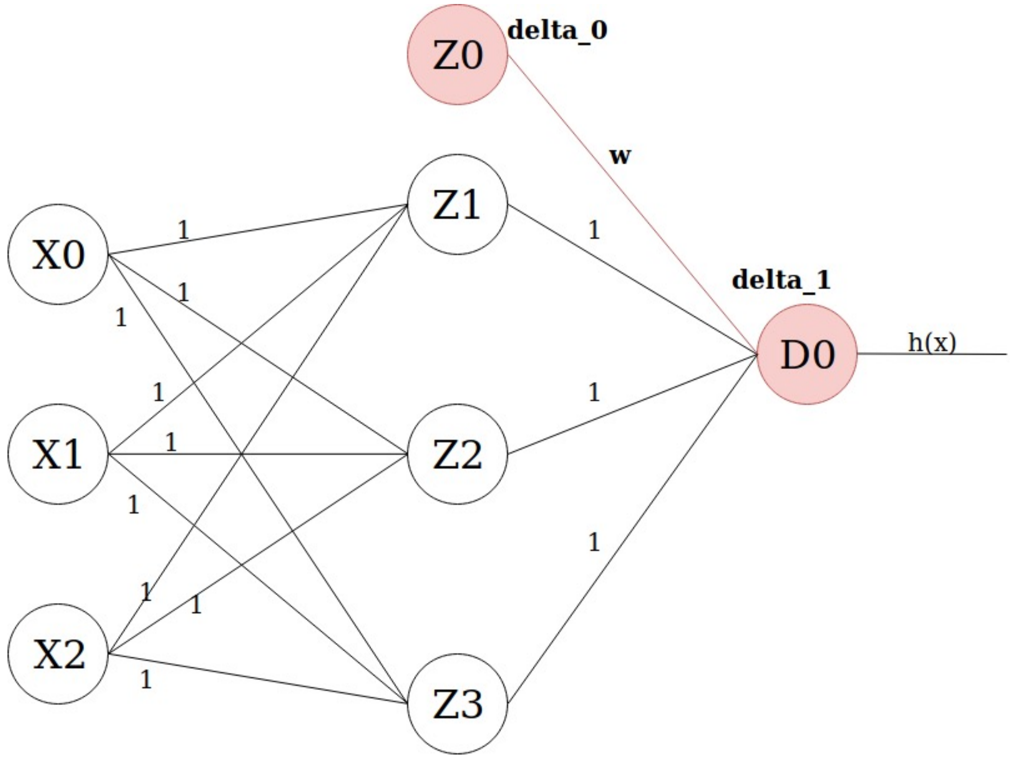
A 2-step process

1. Forward pass - Compute the network output
2. Backward pass – Compute the loss function gradients and update

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Backpropagation in other words

- In order to get the loss of a node (e.g. Z0), we multiply the value of its corresponding f'(z) by the loss of the node it is connected to in the next layer (delta_1), by the weight of the link connecting both nodes.
- We do the delta calculation step at every unit, back-propagating the loss into the neural net, and finding out what loss every node/unit is responsible for.



https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7

On the Key Importance of Error Functions

- The error/loss/cost function reduces all the various good and bad aspects of a possibly complex system down to a single number, a scalar value, which allows candidate solutions to be compared.
- It is important, therefore, that **the function faithfully represent our design goals**.
- If we choose a poor error function and obtain unsatisfactory results, the fault is ours for badly specifying the goal of the search.

Objective Functions for NN

- Regression: A problem where you predict a real-value quantity.
  - Output Layer: One node with a linear activation unit.
  - Loss Function: Quadratic Loss (Mean Squared Error (MSE))
- Classification: Classify an example as belonging to one of K classes
  - Output Layer:
    - One node with a sigmoid activation unit (K=2)
    - K output nodes in a softmax layer (K>2)
  - Loss function: Cross-entropy (i.e. negative log likelihood)

J = E	Forward	Backward
Quadratic	$J = \frac{1}{2}(y - y^*)^2$	$\frac{dJ}{dy} = y - y^*$
Cross Entropy	$J = y^* \log(y) + (1 - y^*) \log(1 - y)$	$\frac{dJ}{dy} = y^* \frac{1}{y} + (1 - y^*) \frac{1}{y - 1}$

Design Issues in ANN

- Number of nodes in input layer
  - One input node per binary/continuous attribute
  - *k* or *log_2k* nodes for each categorical attribute with k values
- Number of nodes in output layer
  - One output for binary class problem
  - *k* or *log_2k* nodes for k-class problem
- Number of nodes in hidden layer
- Initial weights and biases

Characteristics of ANN

- Multilayer ANN are universal approximators but could suffer from ***overfitting*** if the network is too large.
- Gradient descent may converge to ***local minimum***.
- Model building can be very time consuming, but testing can be very fast.
- Can handle redundant attributes because weights are automatically learnt.
- Sensitive to noise in training data.
- Difficult to handle missing attributes.

Tips and Tricks of NN Training

Dataset Should Normally be Split Into

- ***Training set:*** use to update the weights. Records in this set are repeatedly in random order. The weight update equation are applied after a certain number of records.

- ***Validation set:*** use to decide when to stop training only by monitoring the error and to select the best model configuration

- ***Test set:*** use to test the performance of the neural network. It should not be used as part of the neural network development and model selection cycle

Before Starting: Weight Initialization

- Choice of ***initial weight values is important as this decides starting position in weight space.*** That is, how far away from global minimum
  - Aim is to select weight values which produce midrange function signals
  - Select weight values randomly from uniform probability distribution
  - Normalize weight values so number of weighted connections per unit produces midrange function signal
- Try different random initialization to
  - Assess robustness
  - Have more opportunities to find optimal results

Two learning fashion (plus one)

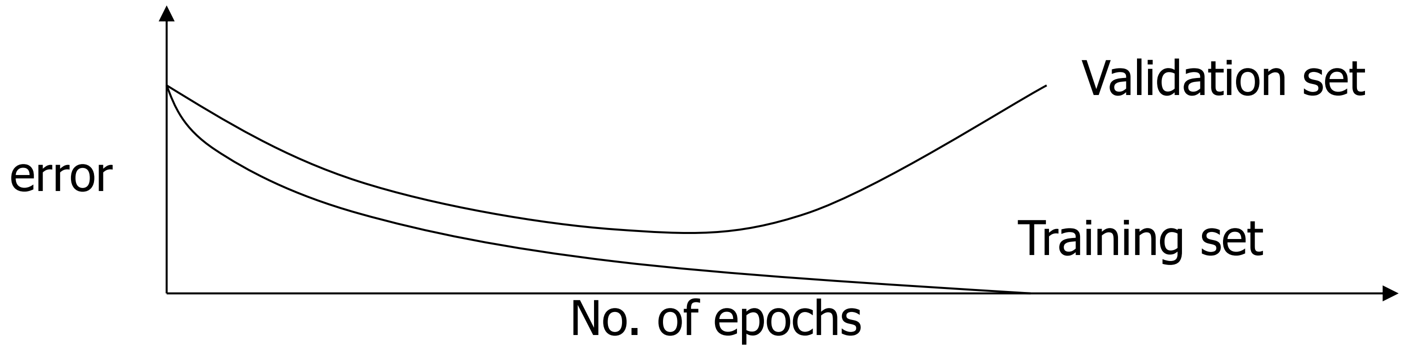
- **Sequential mode** (on-line, stochastic, or per-pattern)
  - Weights updated after each records is presented
  - Many weight updates, can quicker convergence but also make learning less stable
- **Batch mode** (off-line or per-epoch)
  - Weights updated after all records are presented
  - Can be very slow and lead to trapping in early local minima
- **Minibatch mode** (a blend of the two above)
  - Weights updated after a few records (from tens to thousands) are presented
  - Best of both (and good for GPU)

Convergence Criteria

- Learning is obtained by repeatedly supplying training data and adjusting by backpropagation
  - Typically 1 training set presentation = **1 epoch**
- We need a stopping criteria to define convergence
  - Euclidean norm of the gradient vector reaches a sufficiently small value
  - Absolute rate of change in the average squared error per epoch is sufficiently small
  - **Validation for generalization performance: stop when generalization performance reaches a peak**

Early Stopping

- Running too many epochs may **overtrain** the network and result in **overfitting** and perform poorly in generalization
- Keep a hold-out validation set and test accuracy after every epoch. Maintain weights for best performing network on the validation set and stop training when error increases beyond this
- Always let the network run for some epochs before deciding to stop (**patience parameter**), then backtrack to best result



Model Selection

- **Too few hidden units** prevent the network from learning adequately fitting the data and learning the concept.
- **Too many hidden units** leads to overfitting, unless you regularize heavily (e.g. dropout, weight decay, weight penalties)
- Cross validation should be used to determine an appropriate number of hidden units by using the optimal validation error to select the model with optimal number of hidden layers and nodes.

Regularization

- Constrain the learning model to avoid overfitting and help improving generalization.
- Add **penalization terms** to the loss function that *punish* the model for excessive use of resources
  - Limit the **amount of weights** that is used to learn a task
  - Limit the **total activation of neurons** in the network

$$E' = E(y, y^*) + \lambda R(\cdot)$$

Hyperparameter to be chosen in model selection

$R(W_\theta)$ Penalty on parameters

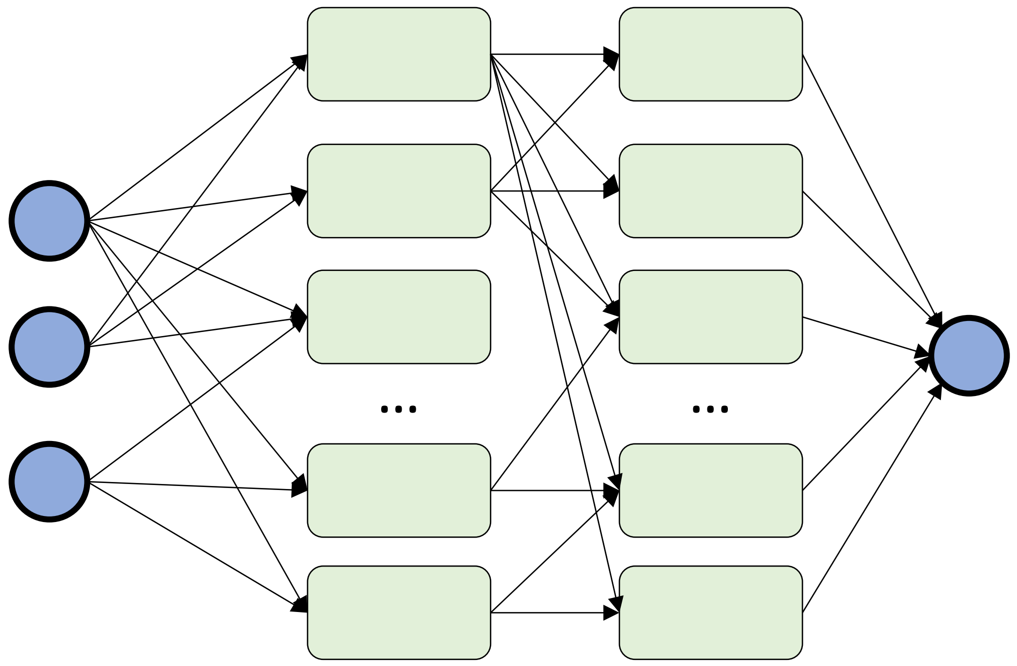
$R(Z)$ Penalty on activations

Common penalty terms (norms)

- 1-norm $||A||_1 = \sum_{ij} |a_{ij}|$
  - Parameters: $R(W_\theta) = ||W_\theta||_1^2$
  - Activations: $R(Z(X)) = ||Z(X)||_1^2$ (Z hidden unit activation)
- 2-norm $||A||_2 = \sqrt{\sum_{ij} a_{ij}^2}$
  - Parameters: $R(W_\theta) = ||W_\theta||_2^2$
  - Activations: $R(Z(X)) = ||Z(X)||_2^2$ (Z hidden unit activation)
- Any p-norm and more…

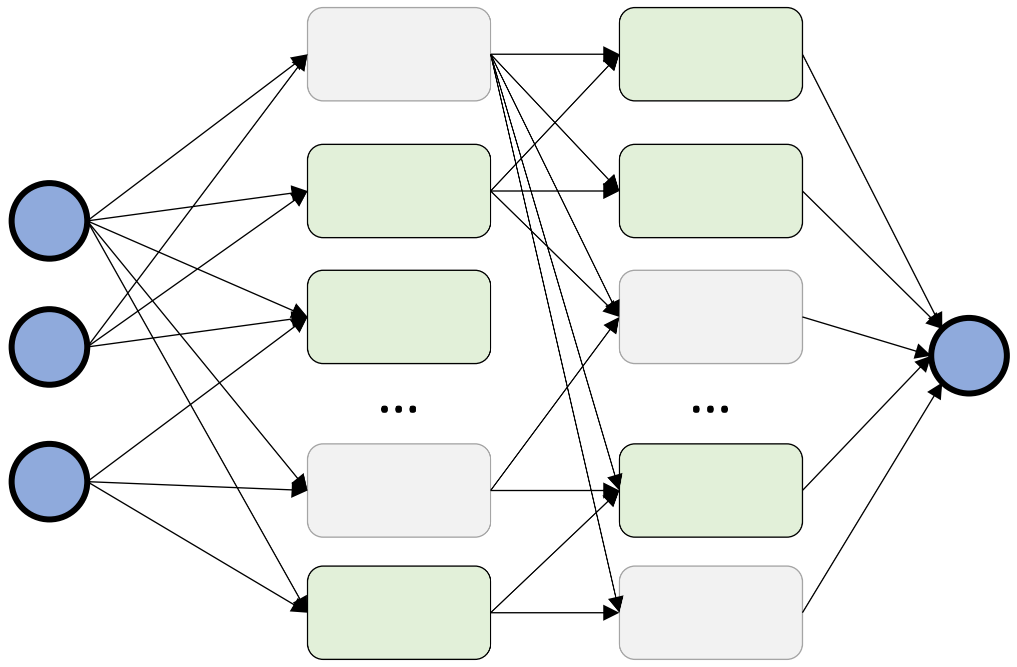
Dropout Regularization

Randomly disconnect units from the network during training



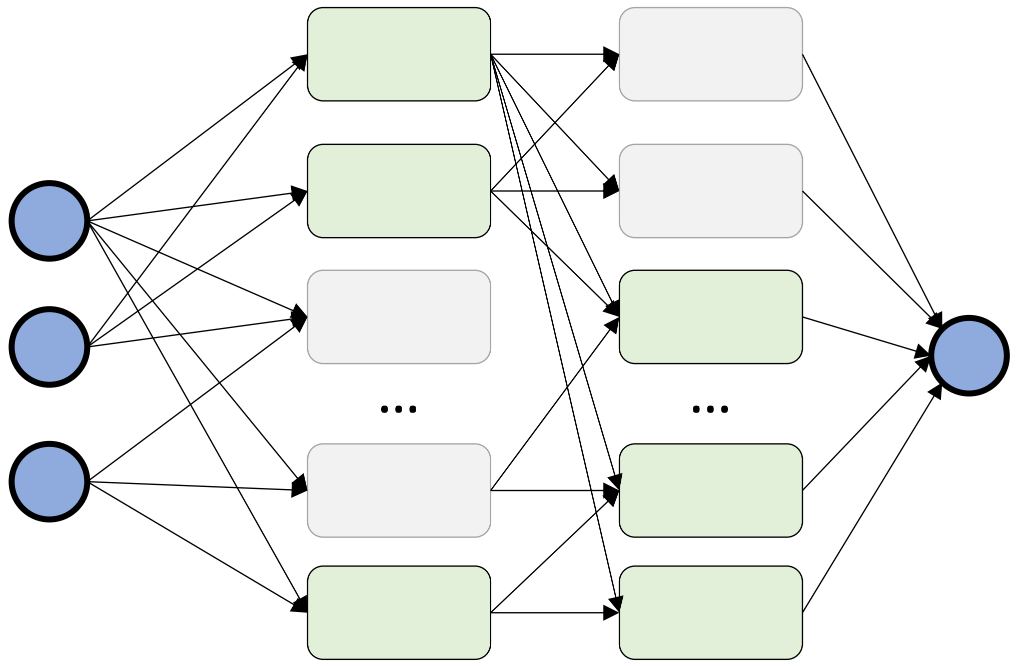
Dropout Regularization

Randomly disconnect units from the network during training



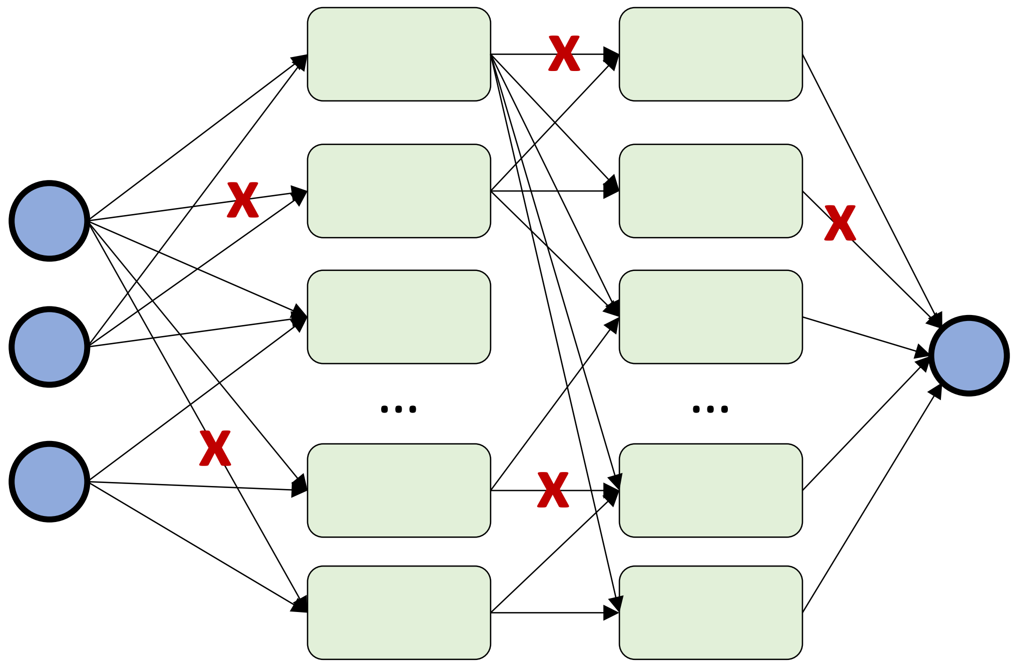
Dropout Regularization

Randomly disconnect units from the network during training



Dropout Regularization

Randomly disconnect units from the network during training



- Regulated by unit **dropping hyperparameter**
- Prevents unit **coadaptation**
- Committee machine effect
- Need to adapt **prediction phase**
- Used at prediction time gives **predictions with confidence intervals**

You can also **drop single connections** (dropconnect)

Momentum

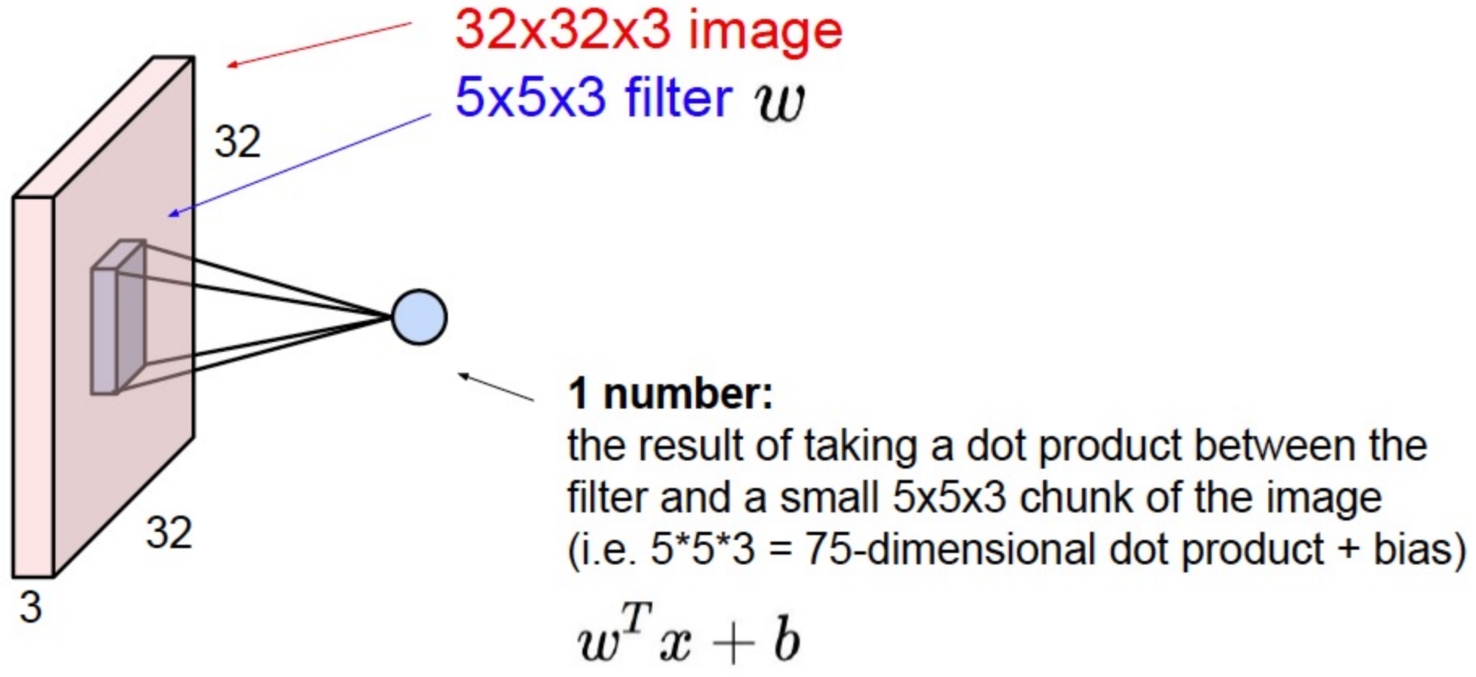
- Adding a term to weight update equation to store an exponentially weight history of previous weights changes
- Reducing problems of instability while increasing the rate of convergence
    - If weight changes tend to have same signs, the momentum term increases and gradient decrease speed up convergence on shallow gradient
    - If weight changes tend have opposing signs, the momentum term decreases and gradient descent slows to reduce oscillations (stabilizes)
    - Can help escape being trapped in local minima

Choosing the Optimization Algorithm

- Standard Stochastic Gradient Descent (SGD)
  - Easy and efficient
  - Difficult to pick up the best learning rate
  - Unstable convergence
  - Often used with **momentum** (exponentially weighted history of previous weights changes)
- RMSprop
  - Adaptive learning rate method (reduces it using a moving average of the squared gradient)
  - Fastens convergence by having quicker gradients when necessary
- Adagrad
  - Like RMSprop with element-wise scaling of the gradient
- **ADAM**
  - **Like Adagrad but adds an exponentially decaying average of past gradients like momentum**

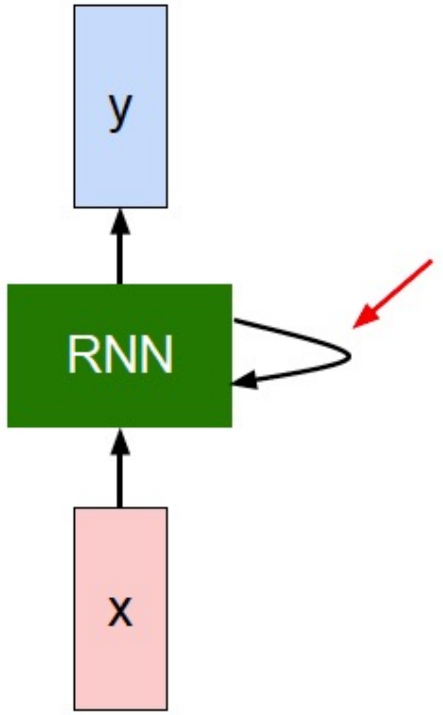
Convolutional Neural Networks

- Are typically applied for the classification of images and time series
- Instead of having only “fully connected” layers adopt “convolutional layers”


32x32x3 image
5x5x3 filter w
32
32
3
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)
$w^T x + b$

Recurrent Neural Network

- Are typically applied in natural language processing (NLP).



Key idea: RNNs have an "internal state" that is updated as a sequence is processed

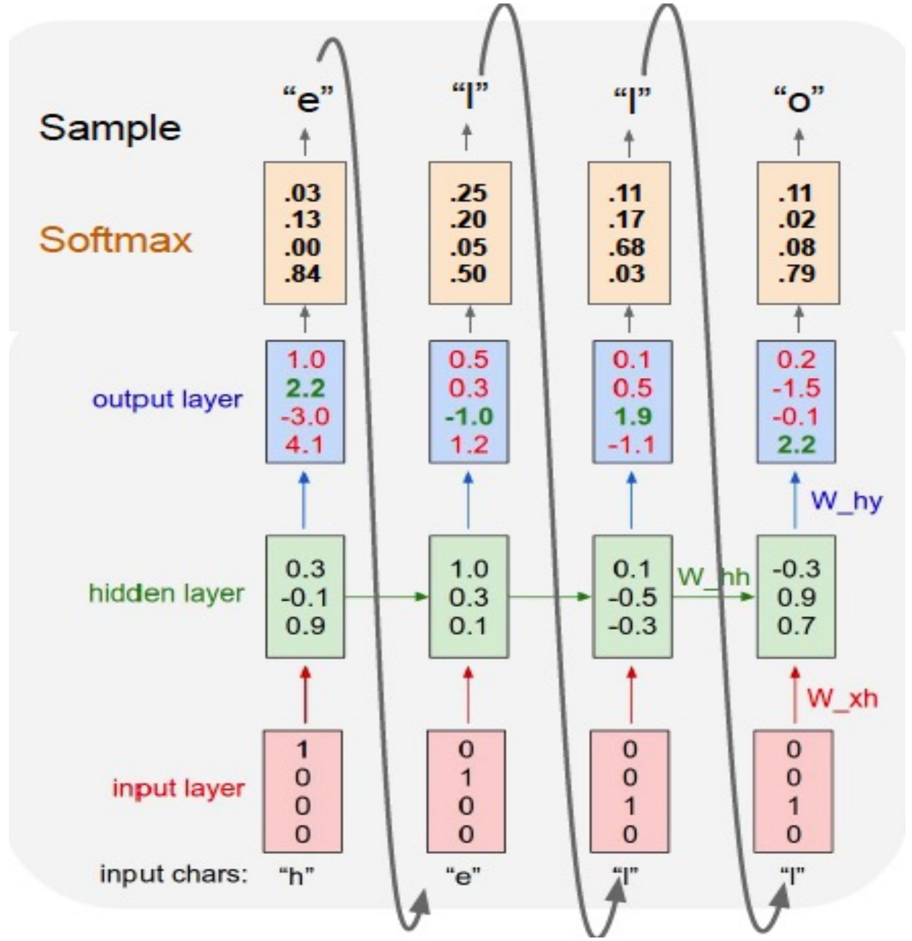
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

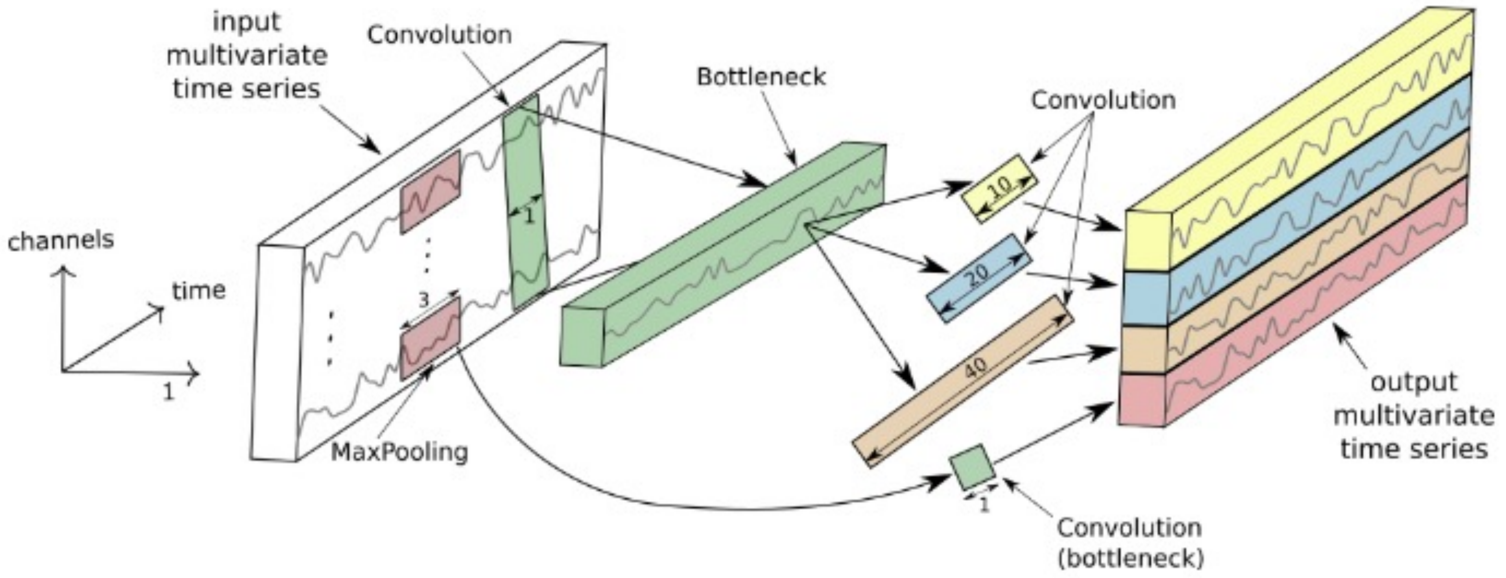
new state

some function with parameters W

old state

input vector at some time step





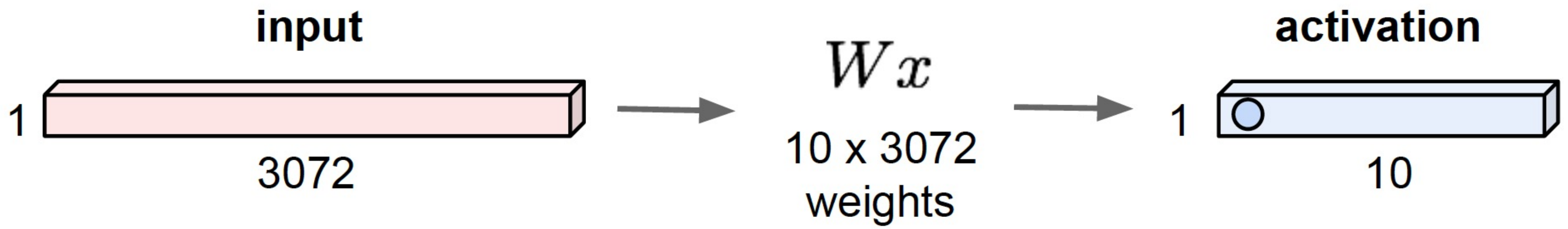
Convolutional Neural Network

Slides edited from Stanford

http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture09.pdf

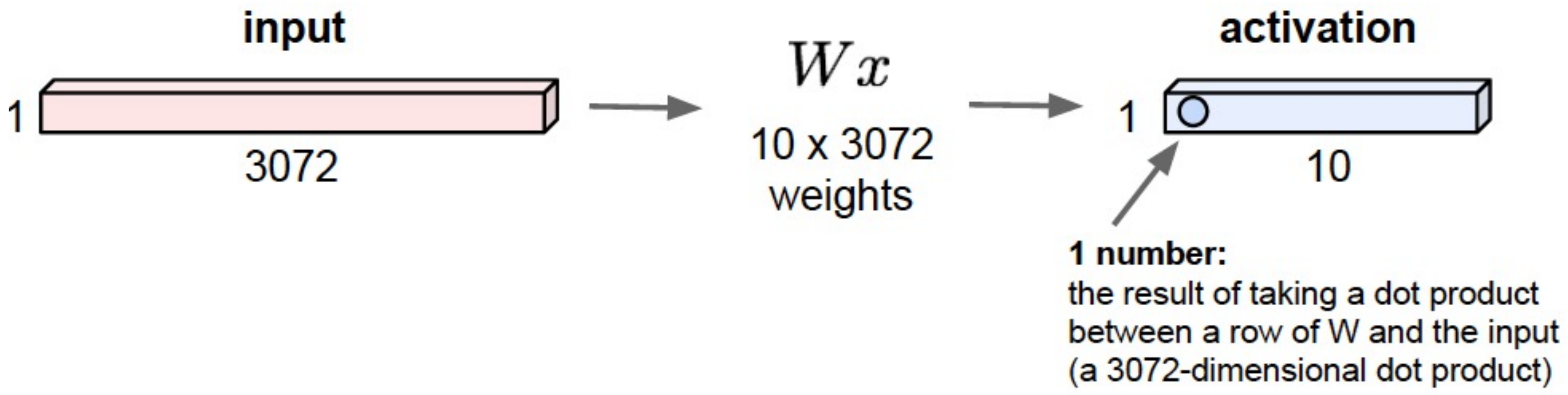
Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



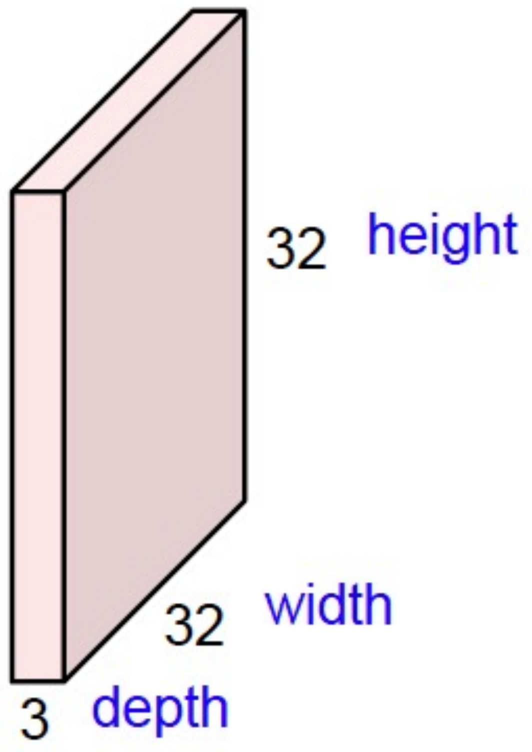
Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1


input
1
3072
Wx
10 x 3072
weights
activation
1
10
1 number:
the result of taking a dot product
between a row of W and the input
(a 3072-dimensional dot product)

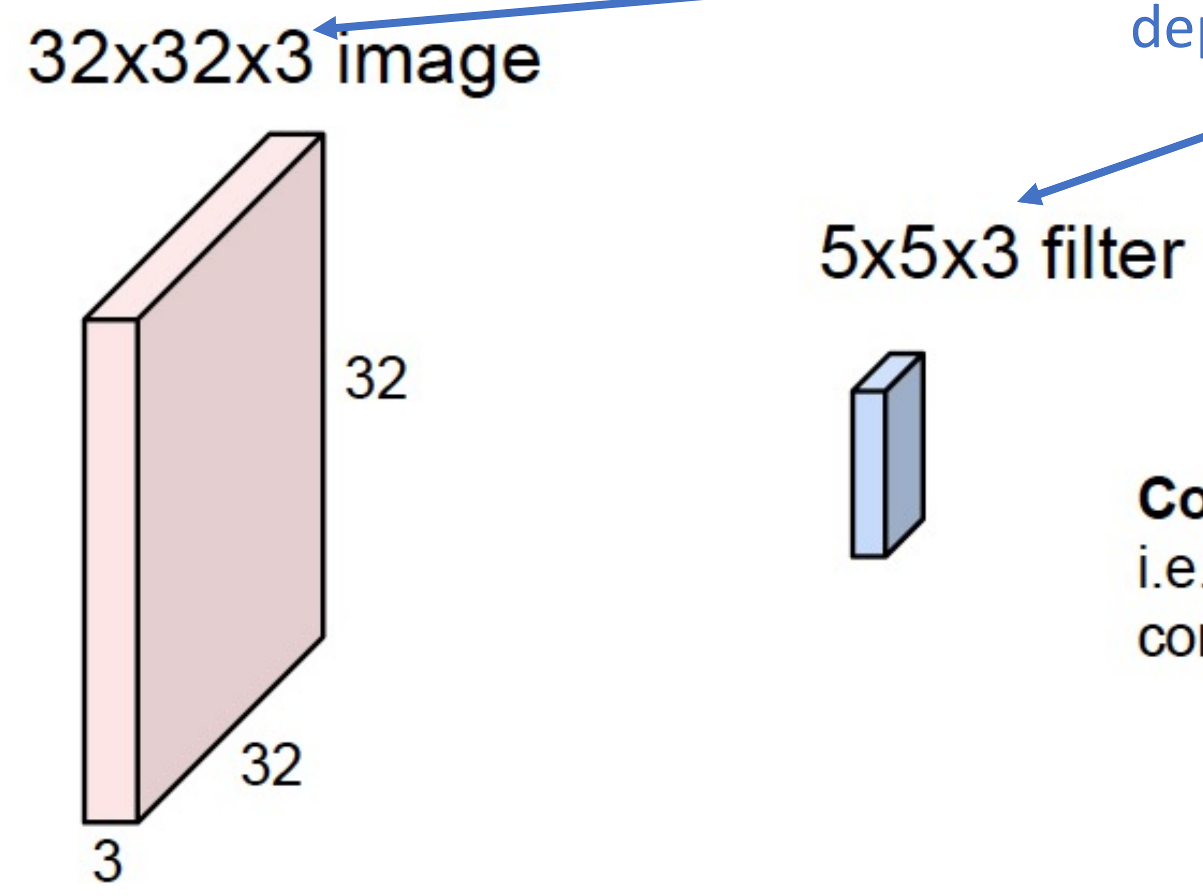
Convolution Layer

32x32x3 image -> preserve spatial structure



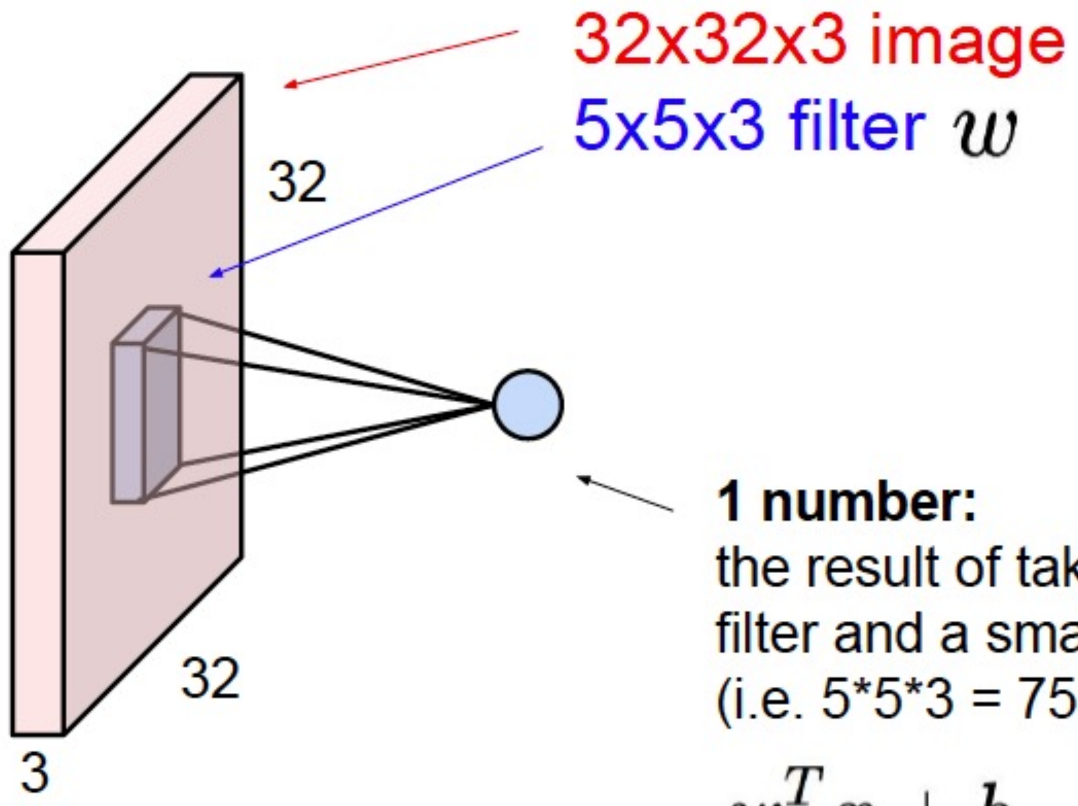
Convolution Layer

Filters always extend the full depth of the input volume



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

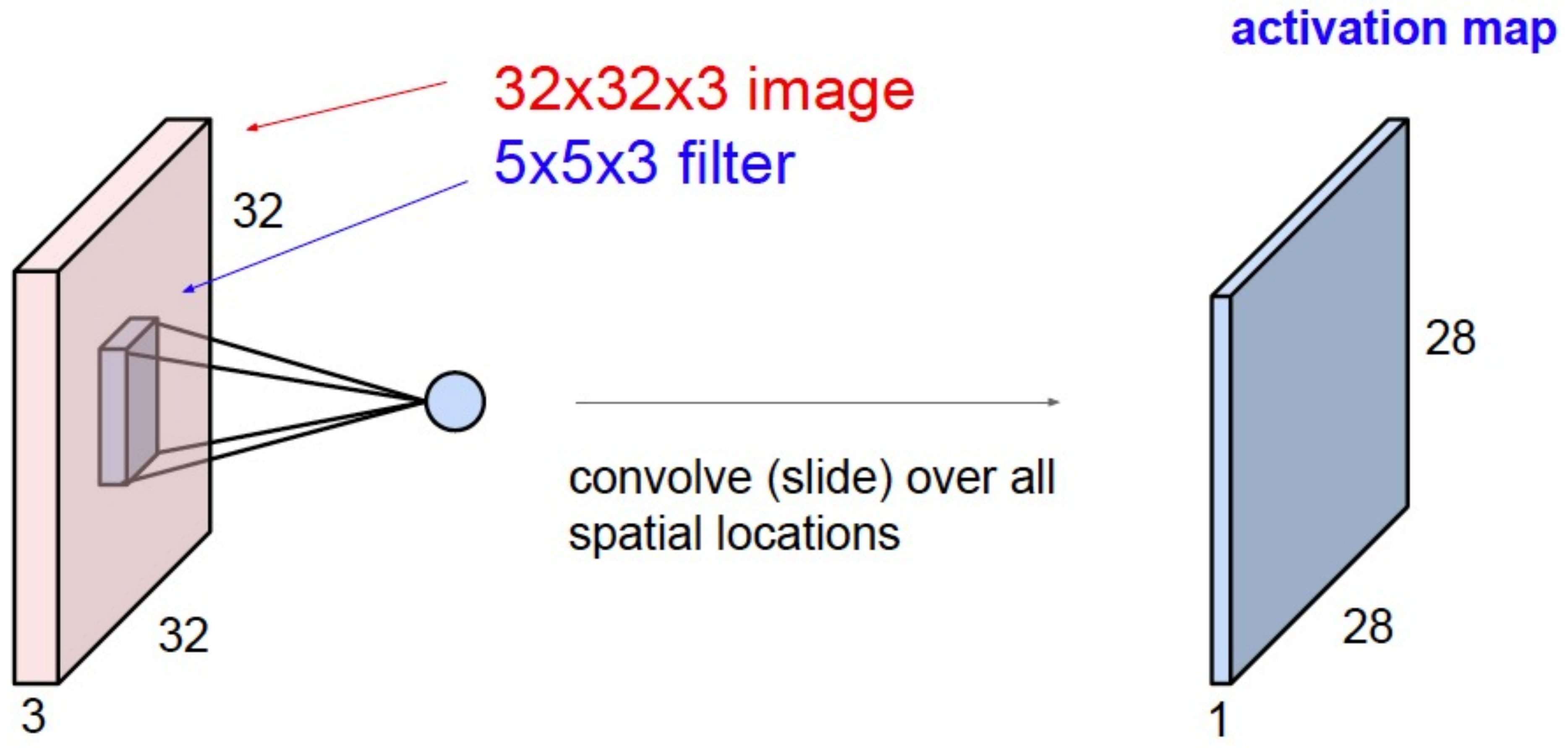
Convolution Layer



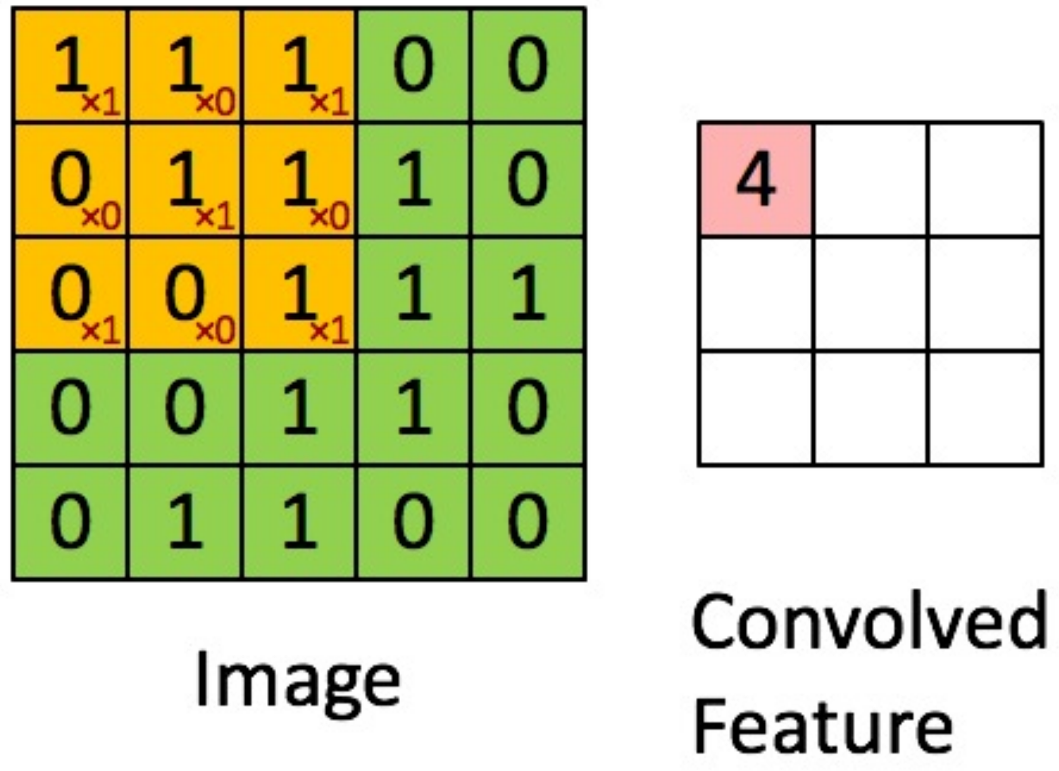
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

Convolution Layer



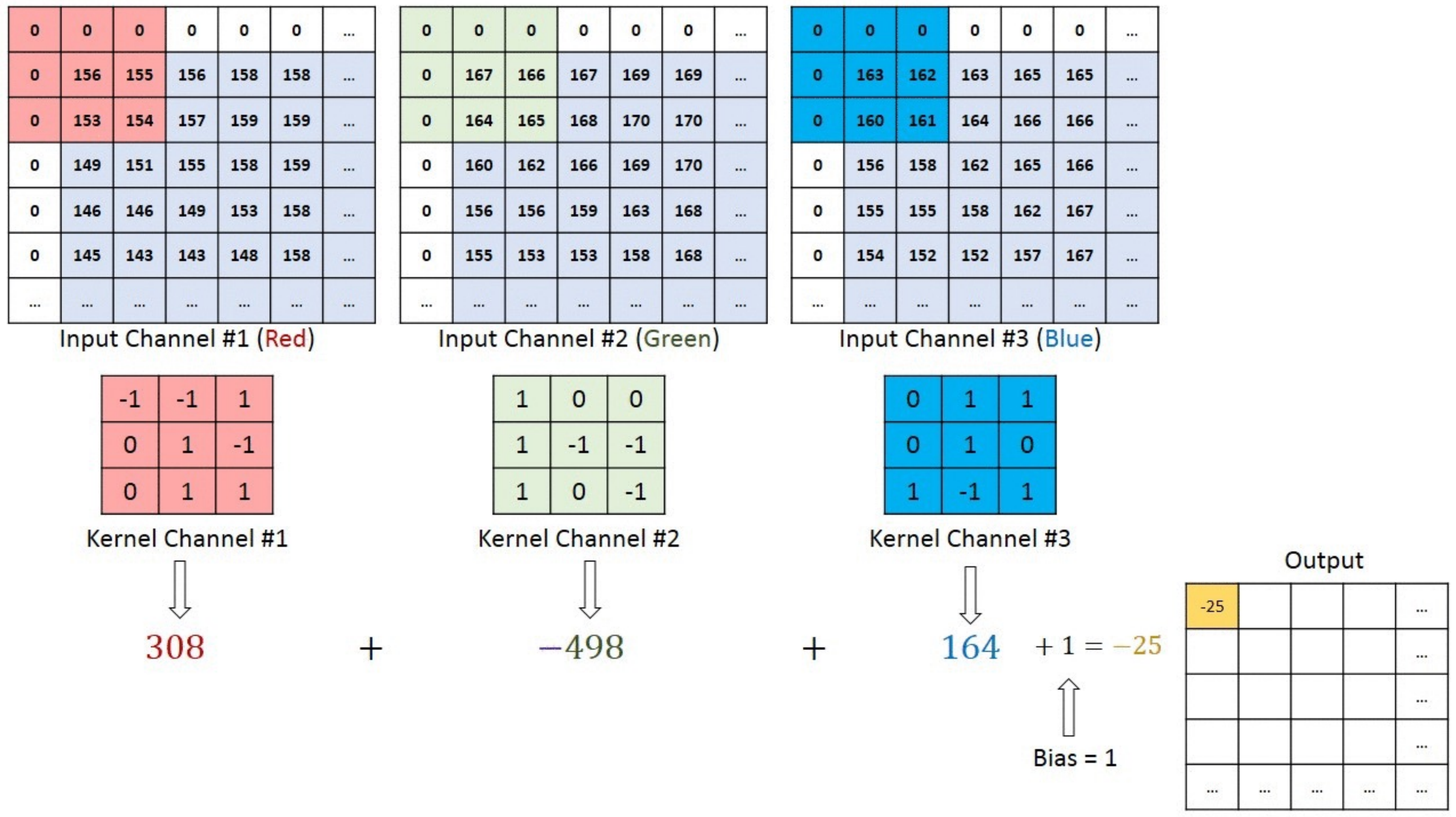
Convolution Layer



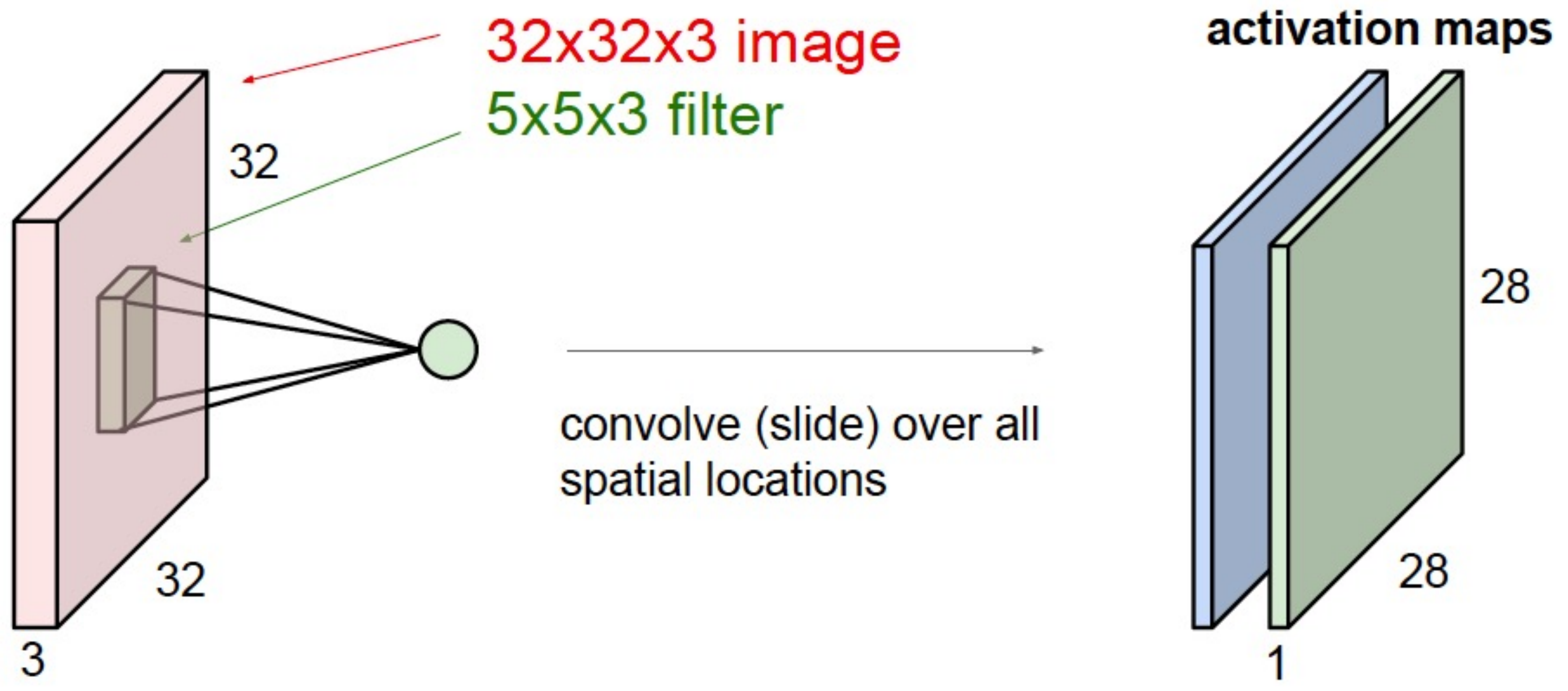
1	0	1
0	1	0
1	0	1

Convolution Kernel

Convolution Layer

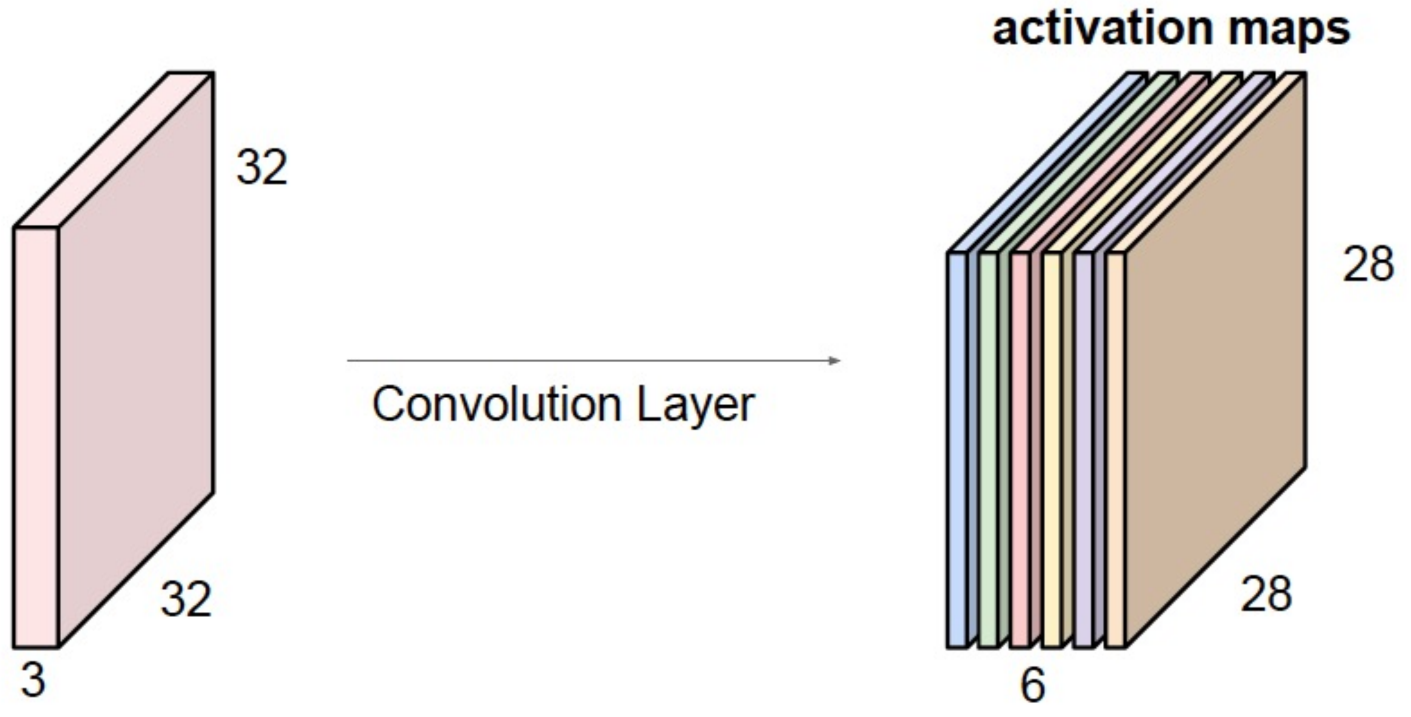


Convolution Layer



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!

Convolutional Neural Network

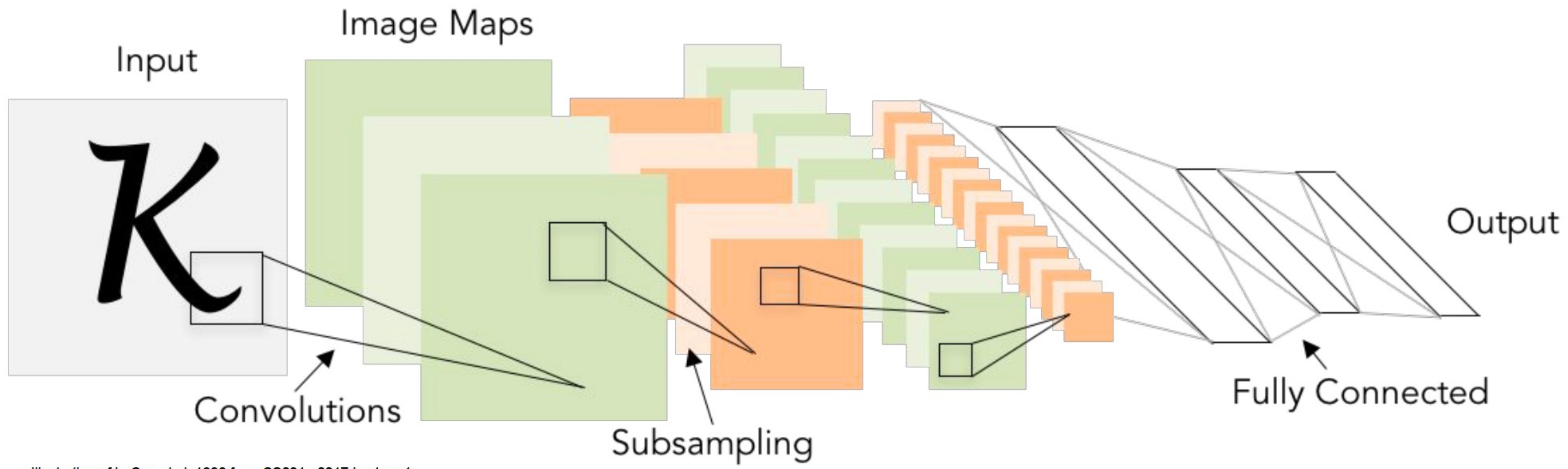
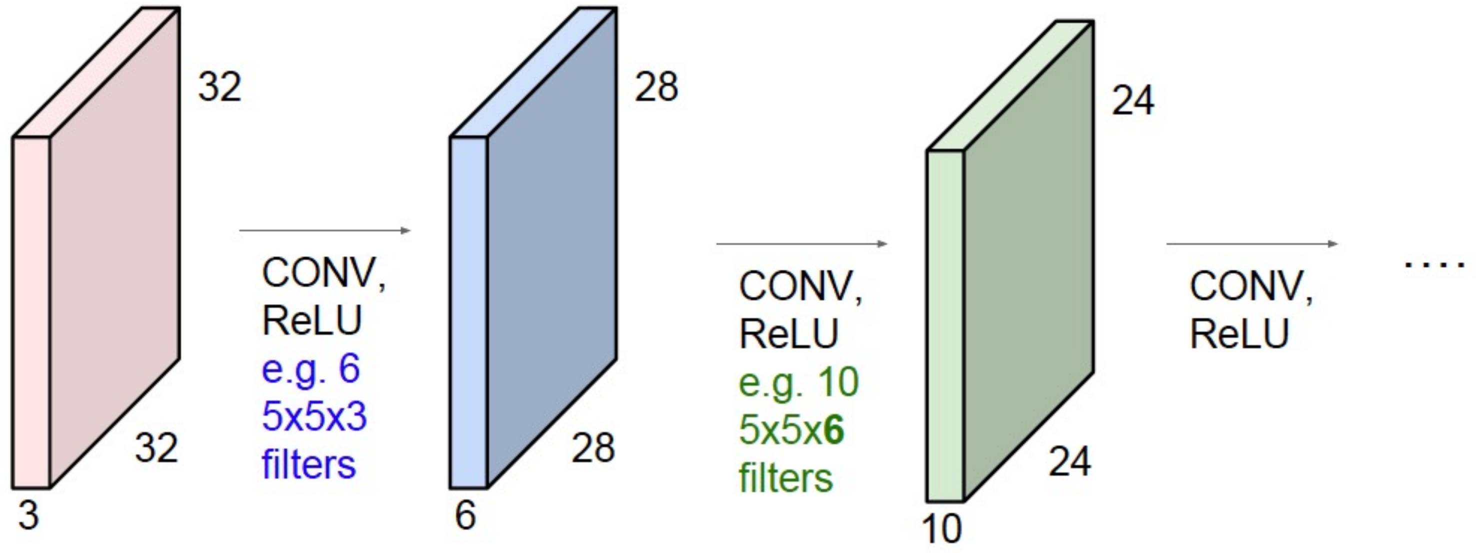


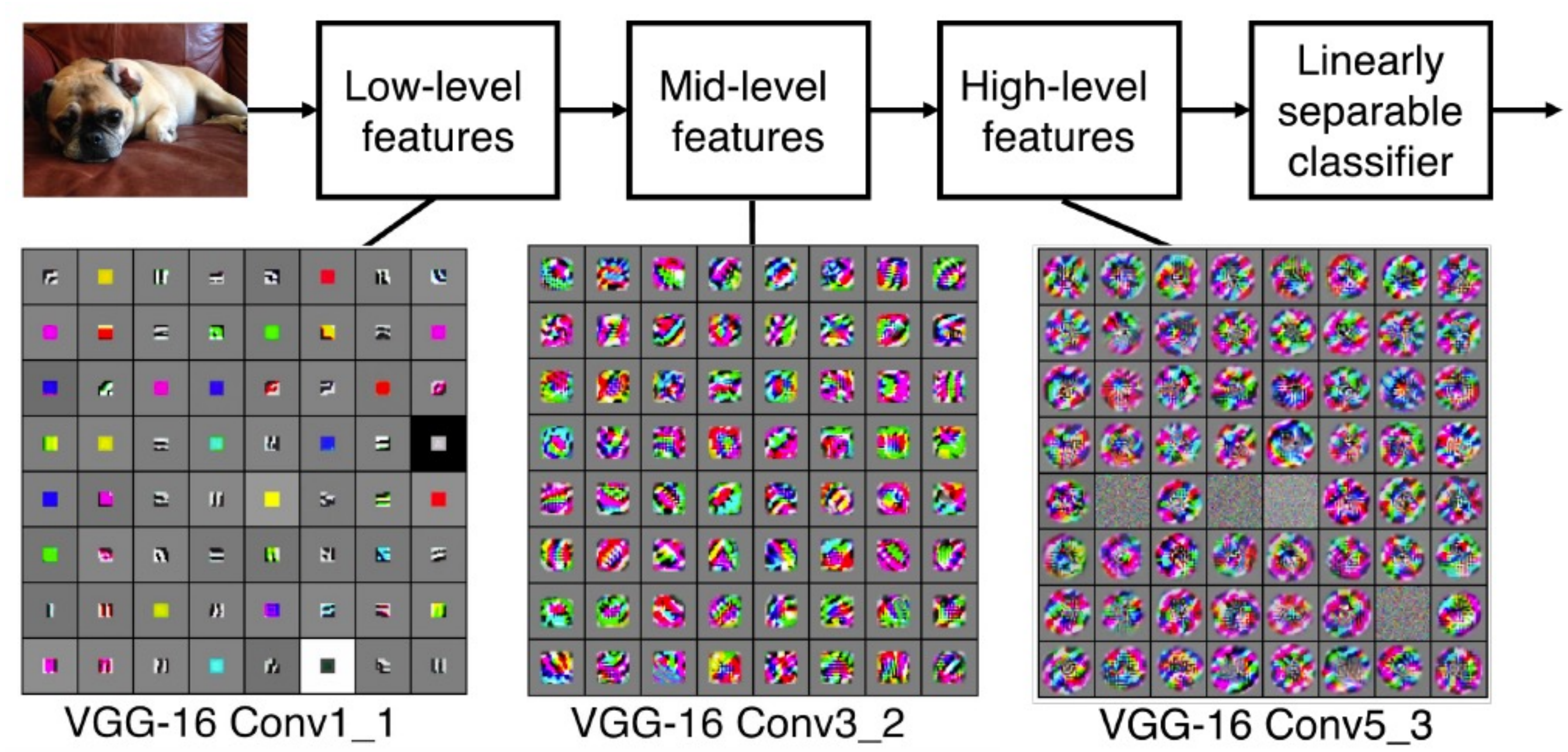
Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Convolutional Neural Network

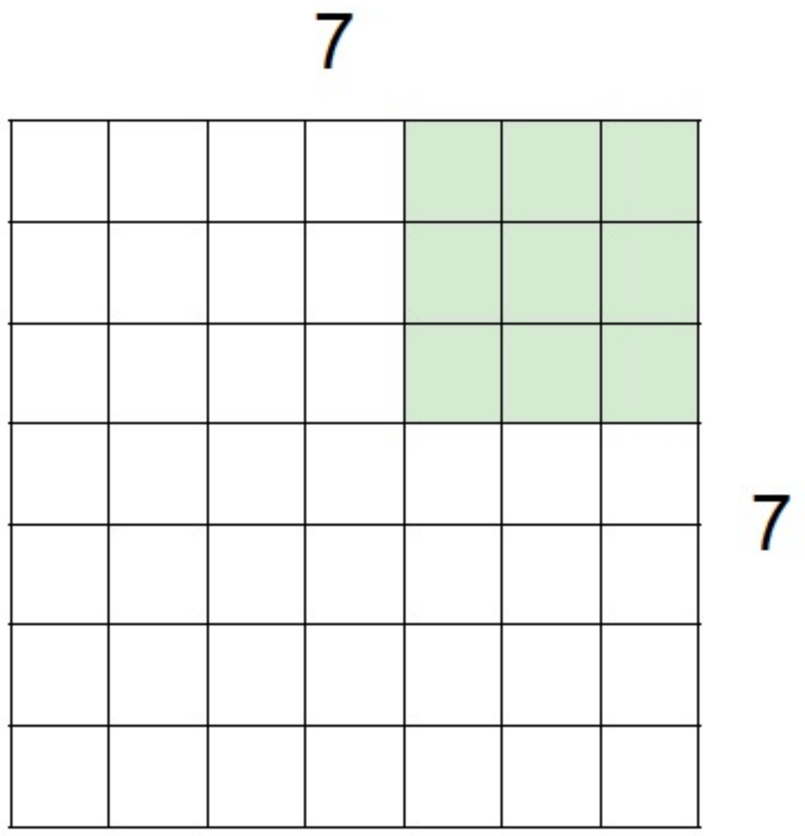
- CNN is a sequence of Conv Layers, interspersed with activation functions.
- CNN shrinks volumes spatially.
- E.g. 32x32 input convolved repeatedly with 5x5 filters! (32 -> 28 -> 24 ...).
- Shrinking too fast is not good, doesn't work well.



CNN for Image Classification



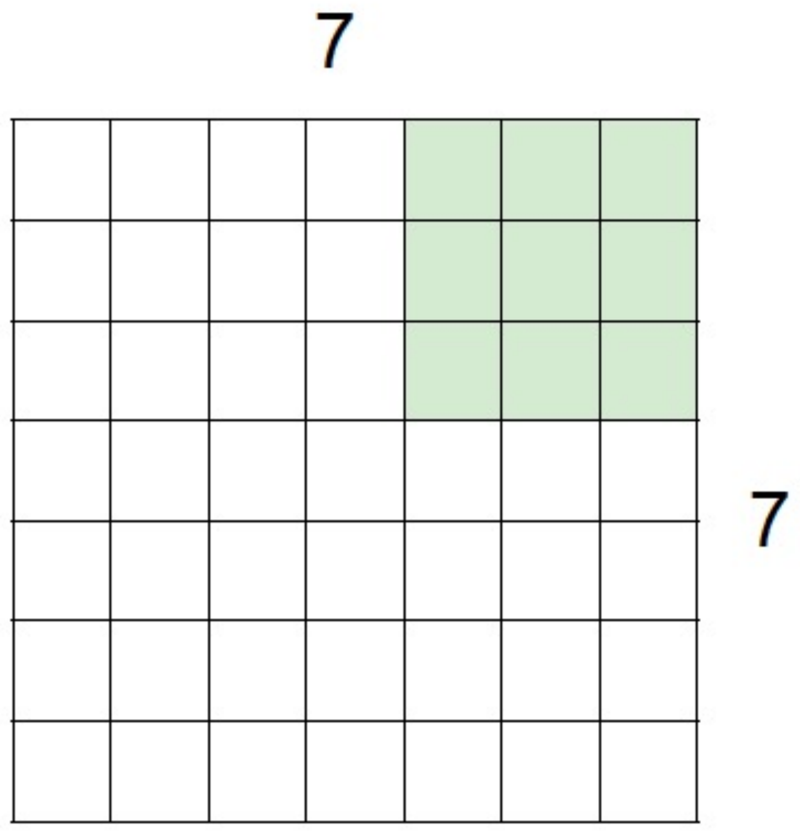
Stride



7x7 input (spatially)
assume 3x3 filter

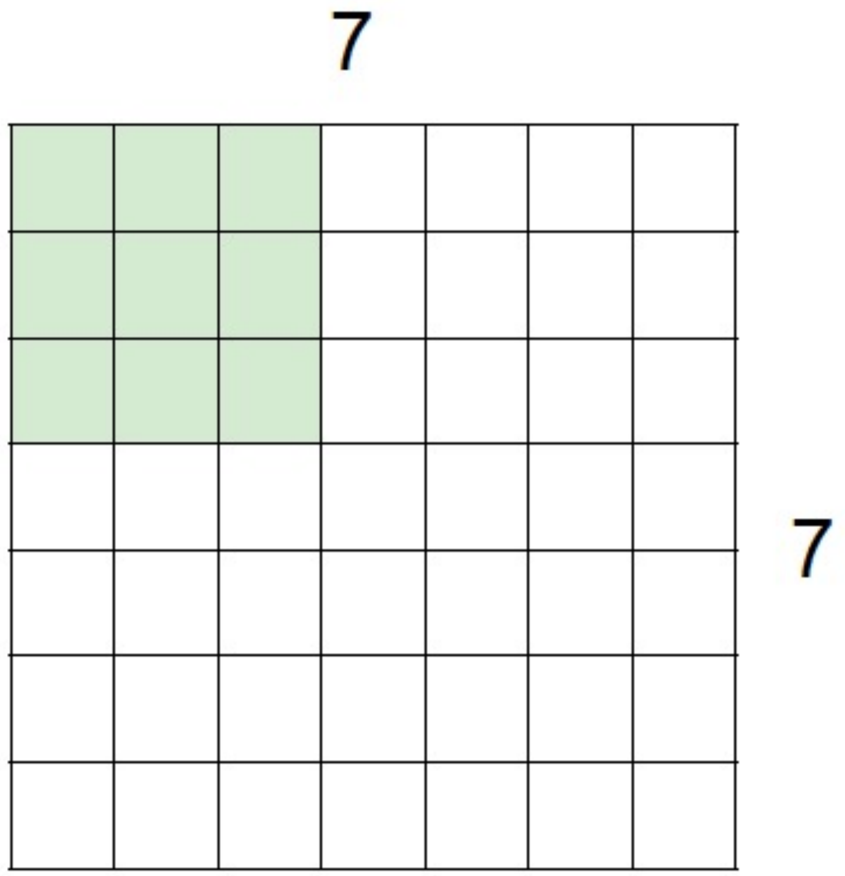
=> 5x5 output

Stride



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

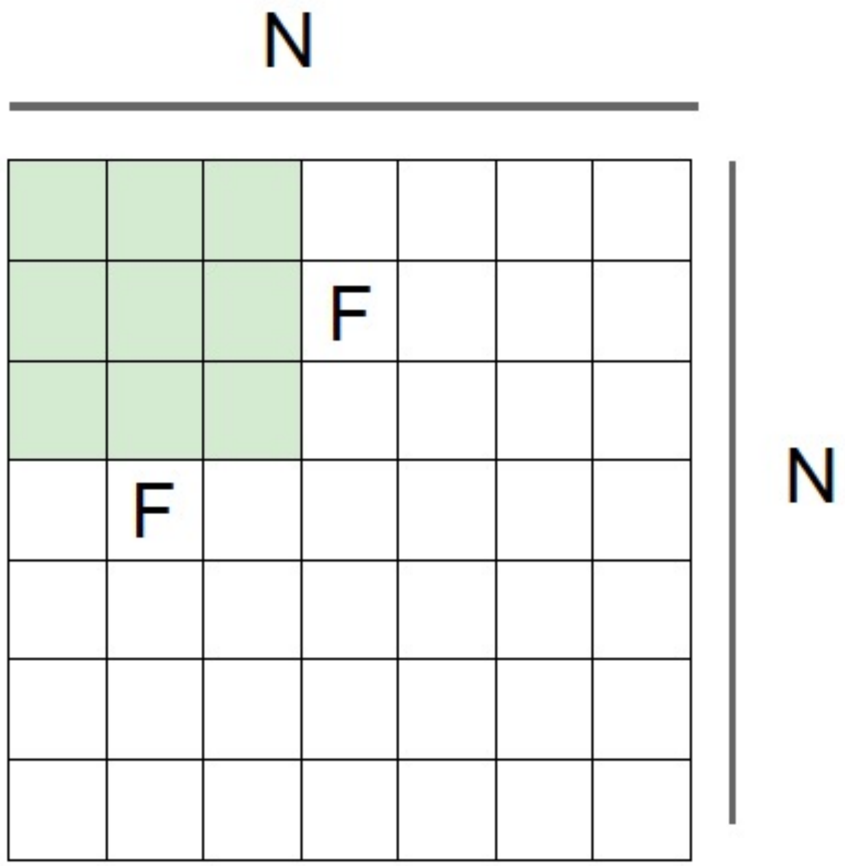
Stride



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

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

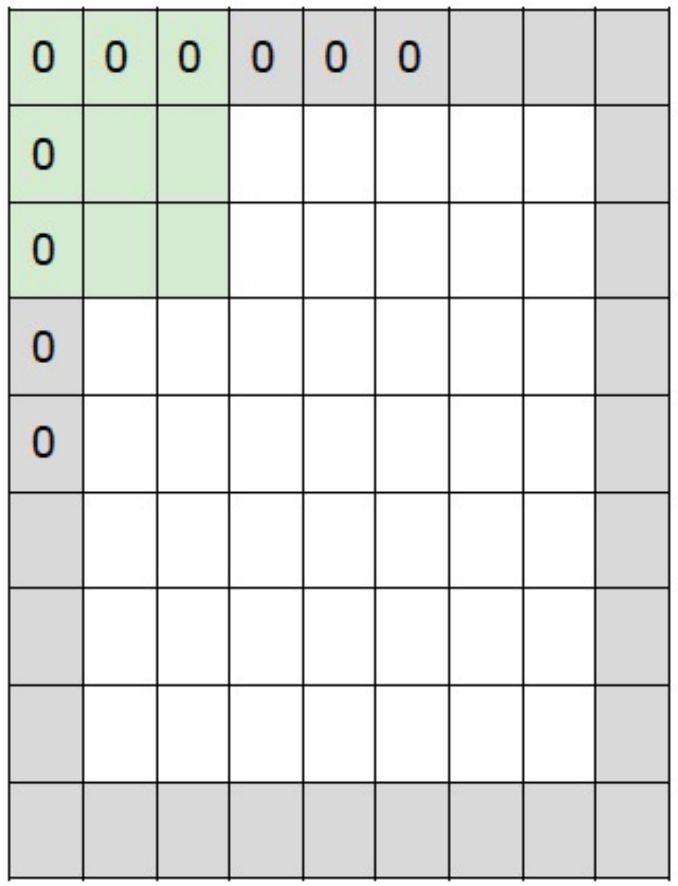
Stride



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 :\

Padding



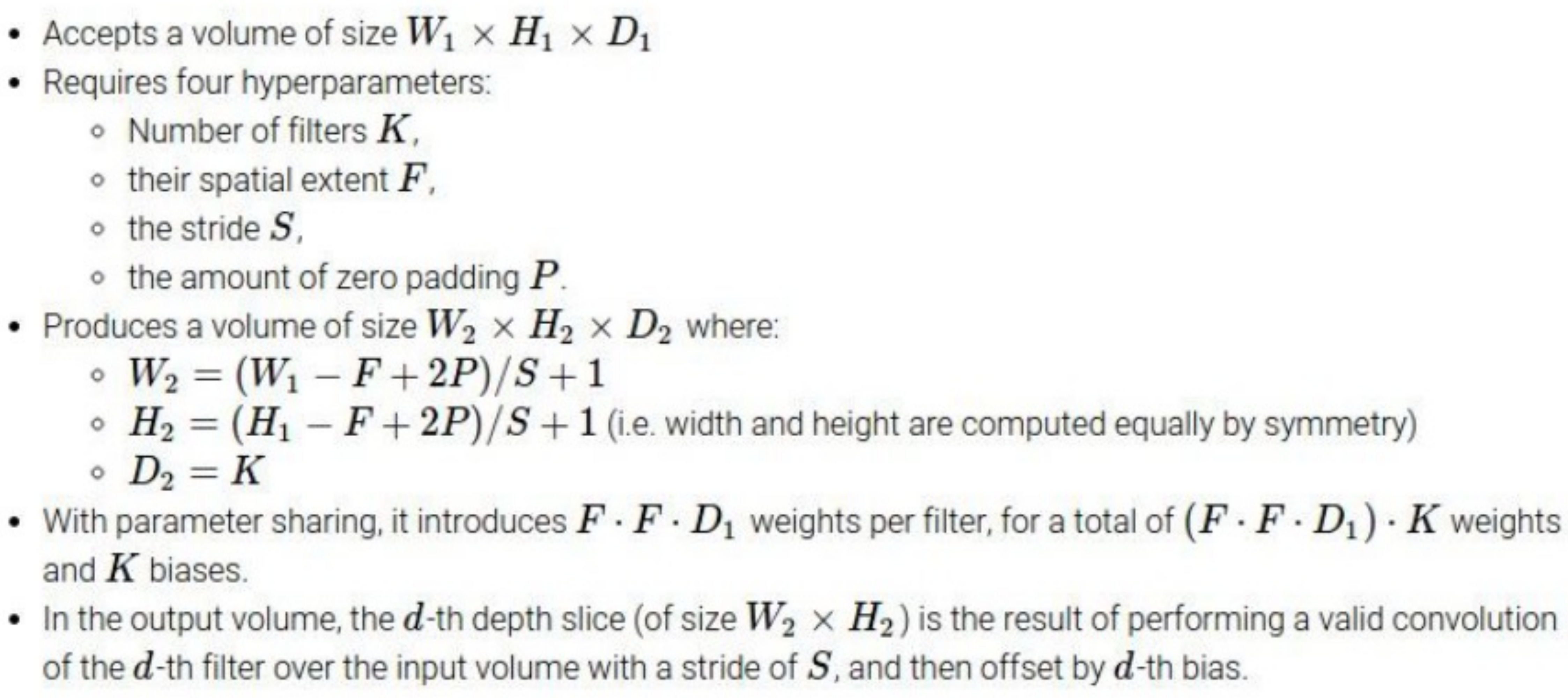
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

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)

- F = 3 => zero pad with 1 pixel
- F = 5 => zero pad with 2 pixel
- F = 7 => zero pad with 3 pixel

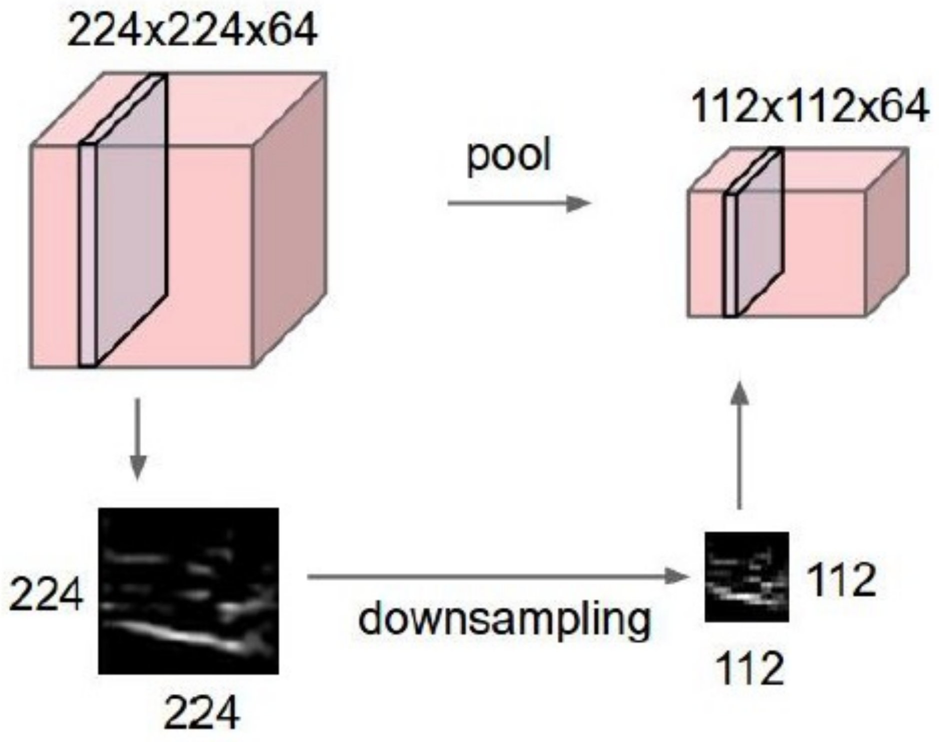
Convolution Summary



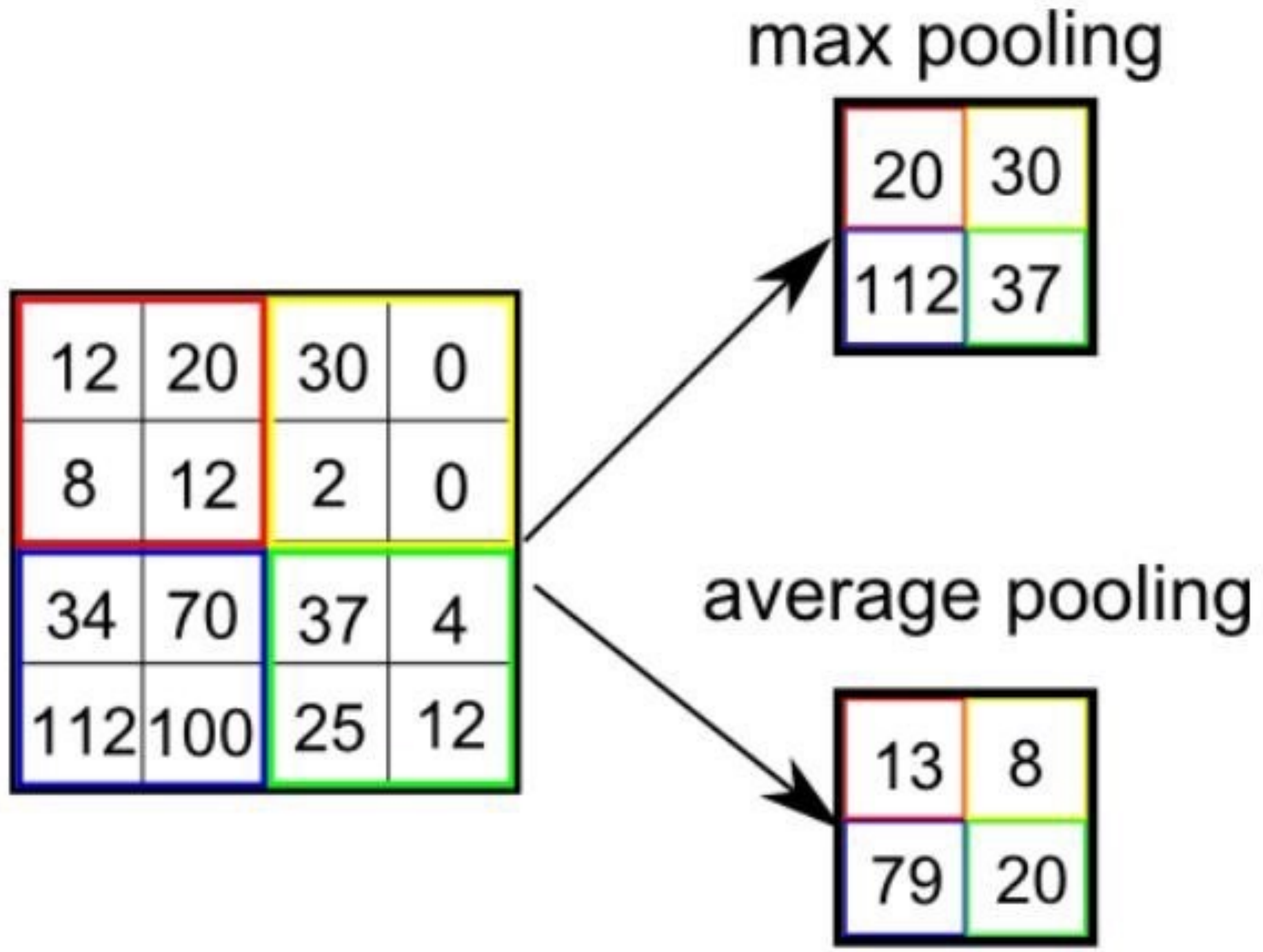
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

Pooling Layer

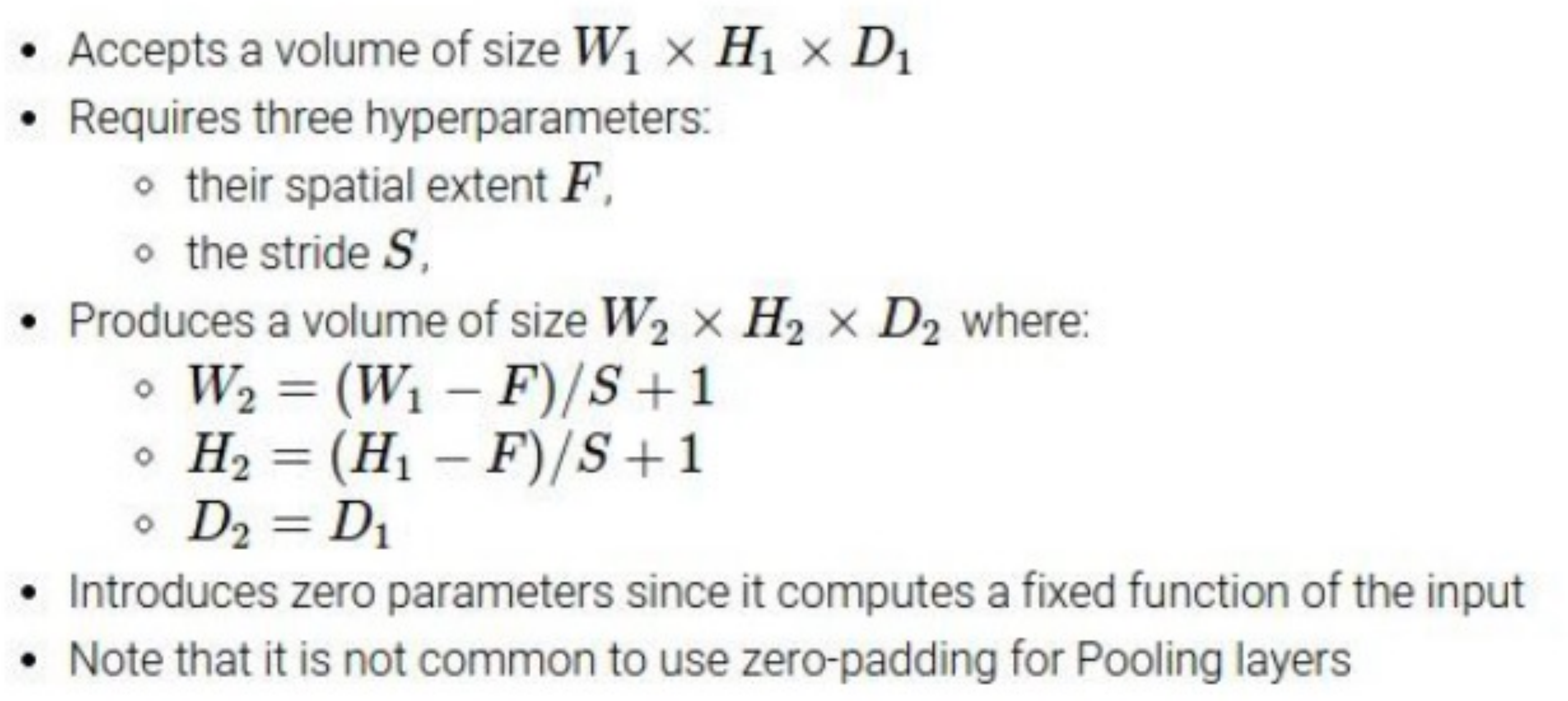
- Makes the representations smaller and more manageable
- Operates over each activation map independently



MaxPooling and AvgPoling

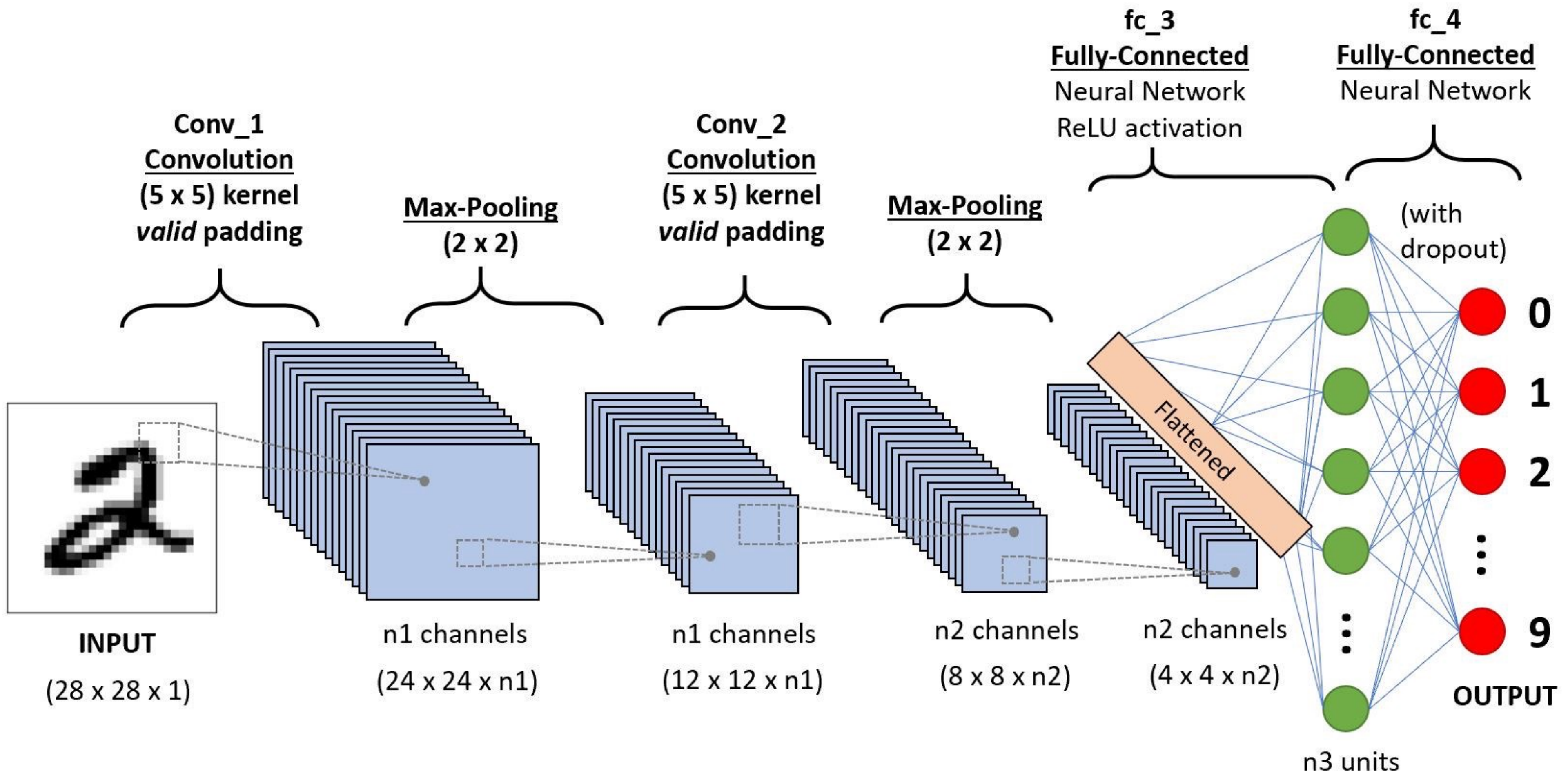


Pooling Summary



- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

Example of CNN

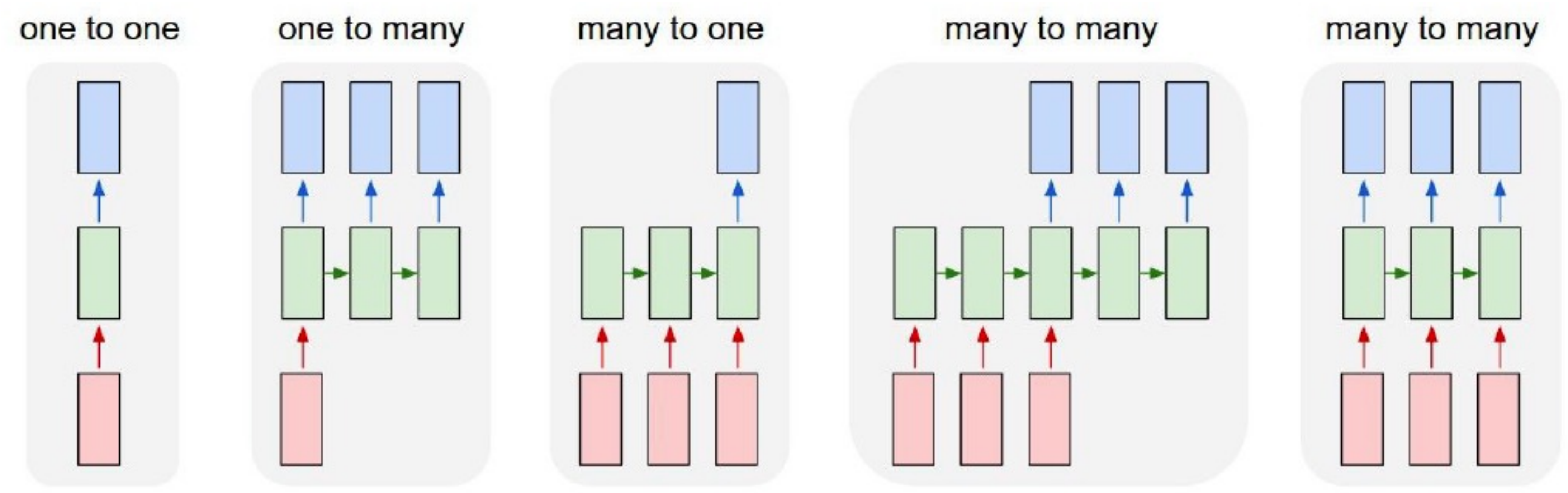


Recurrent Neural Network

Slides edited from Stanford

http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture10.pdf

Types of Recurrent Neural Networks



Vanilla NN

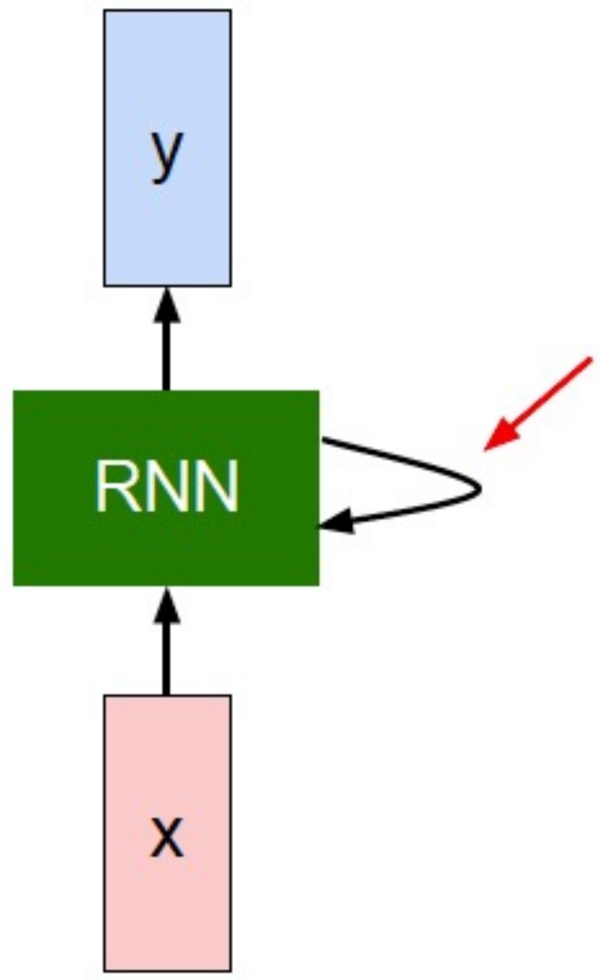
Image -->
Sequence of Words
Image Captioning

Sequence of Words -->
Sentiment
Sentiment Classification
TS Classification

Sequence of Words -->
Sequence of Words
Machine Translation

Video Classification

Recurrent Neural Network - RNN

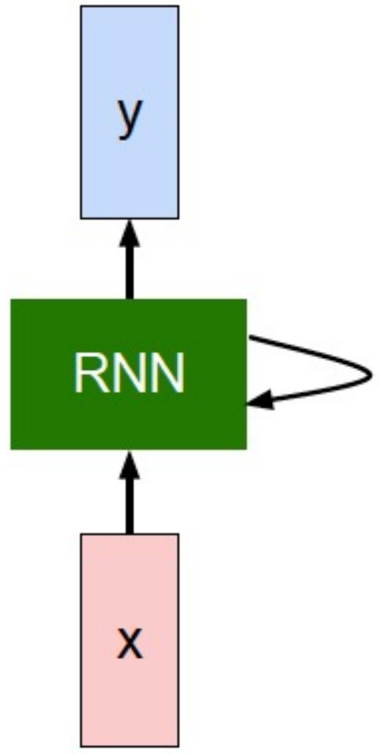


Key idea: RNNs have an "internal state" that is updated as a sequence is processed

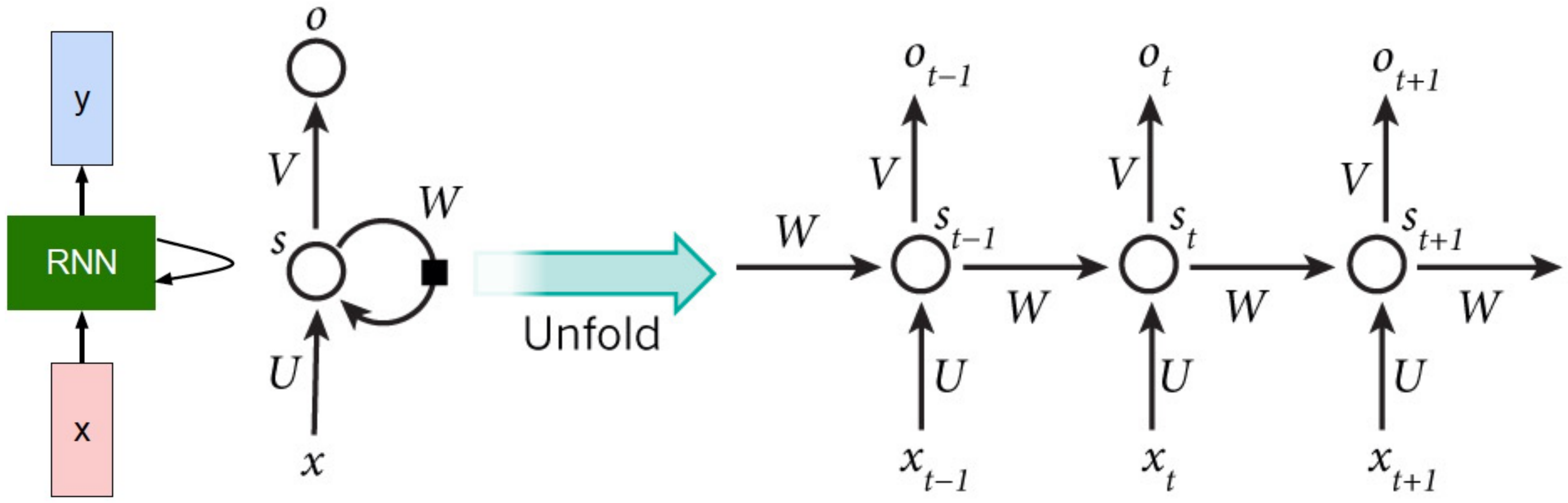
Recurrent Neural Network - RNN

- We can process a sequence of vectors ***x*** by applying a ***recurrence formula*** at every time step:

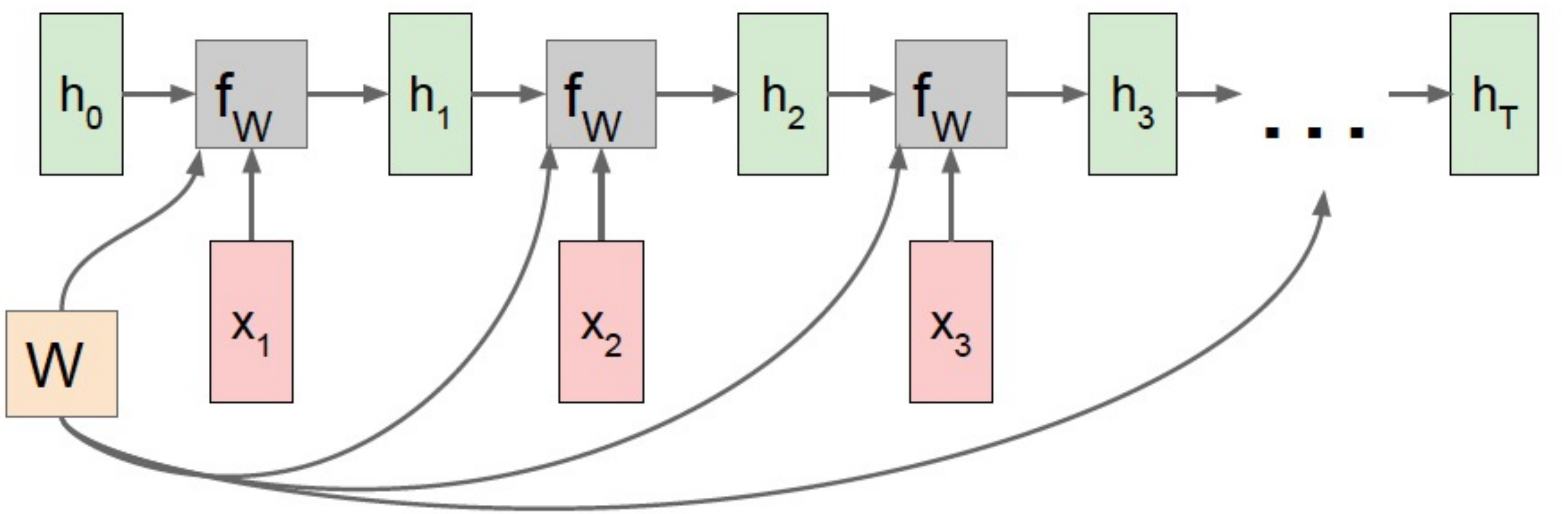
$$\underset{\text{new state}}{\boxed{h_t}} = \underset{\substack{\text{some function}\\ \text{with parameters W}}}{\boxed{f_W}}(\underset{\text{old state}}{\boxed{h_{t-1}}}, \underset{\substack{\text{input vector at}\\ \text{some time step}}}{\boxed{x_t}})$$



Unfolded RNN

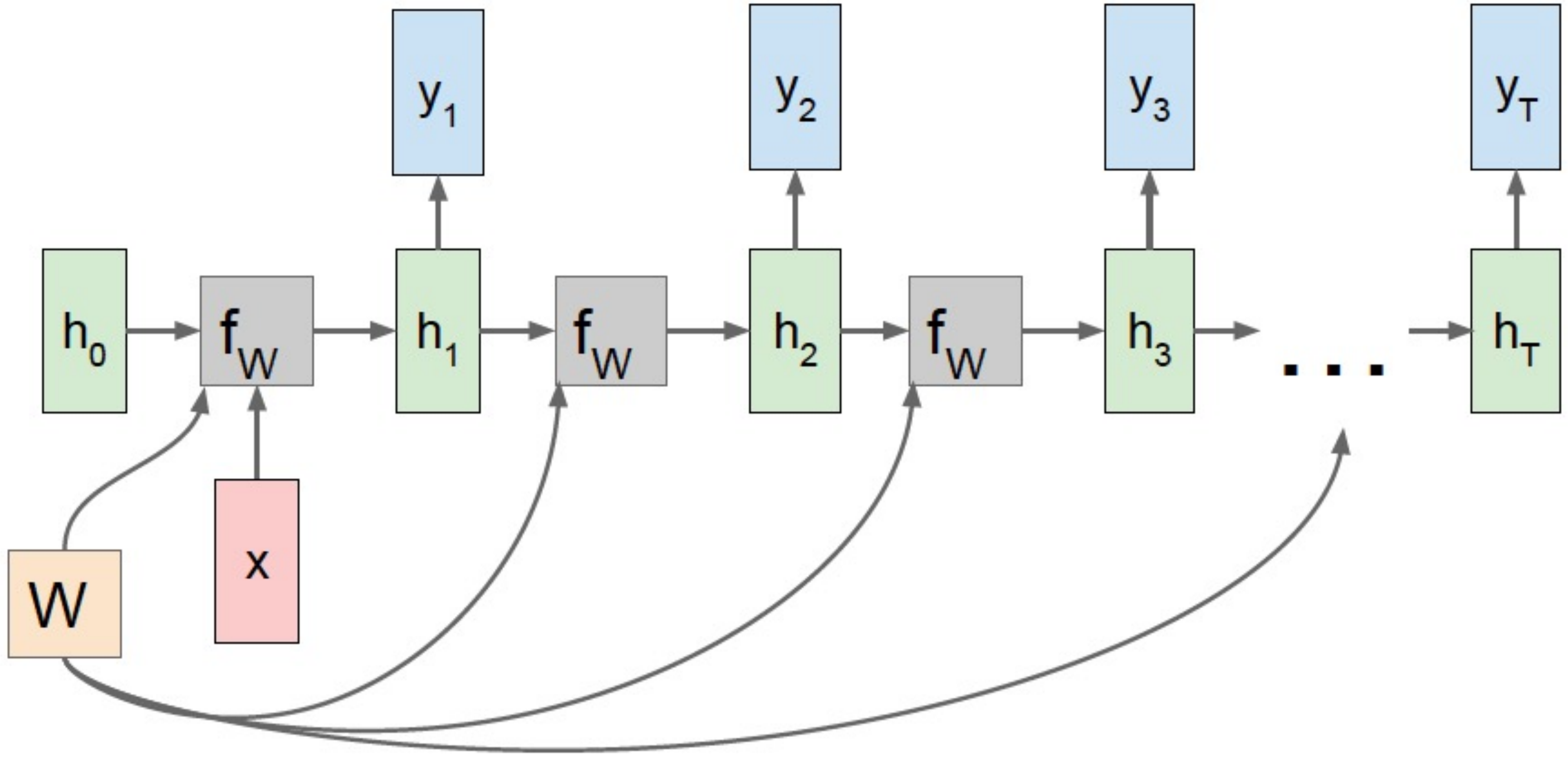


RNN: Computational Graph

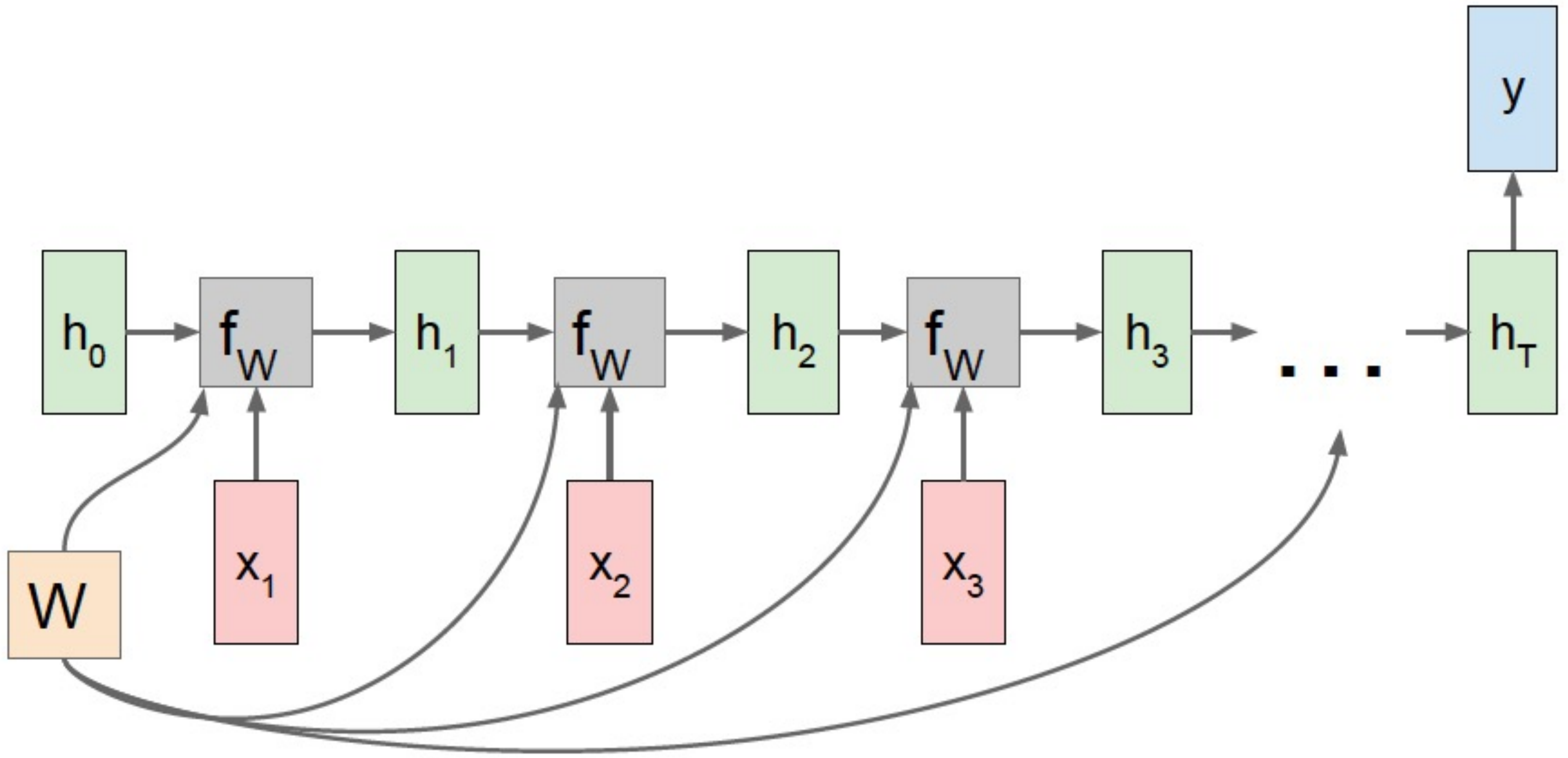


Reminder: Re-use the same weight matrix at every time-step

RNN: Computational Graph: Many to Many



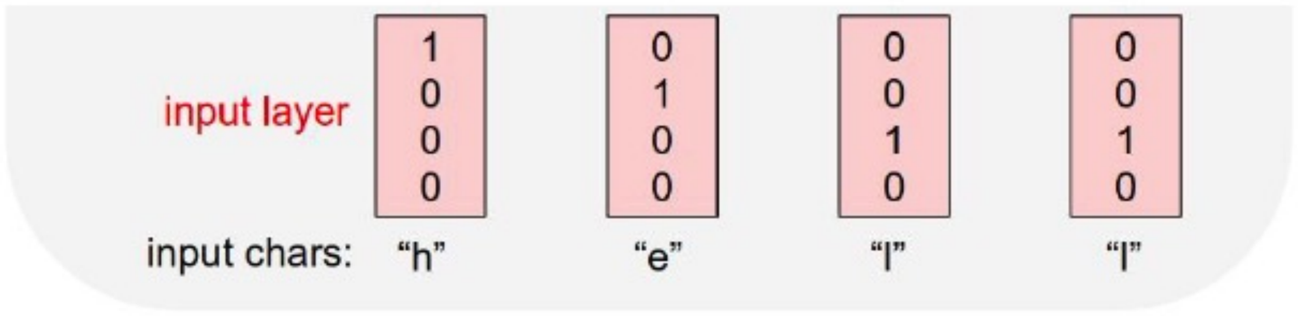
RNN: Computational Graph: Many to One



RNN: Example Training

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”



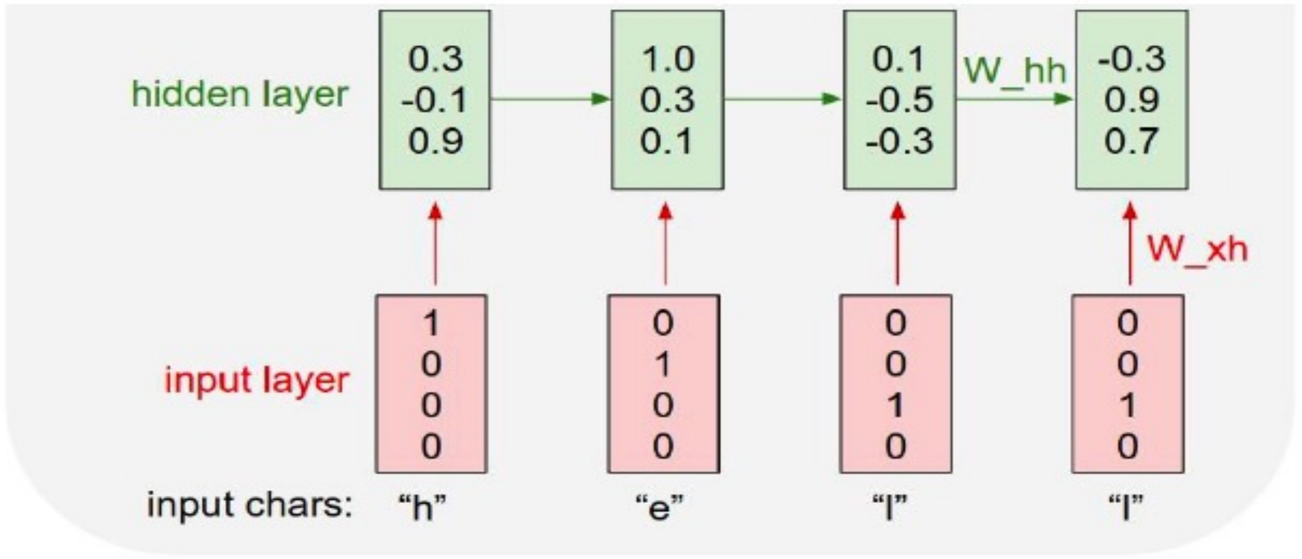
RNN: Example Training

Example: Character-level Language Model

Vocabulary: [h,e,l,o]

Example training sequence: **“hello”**

$$\boxed{h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)}$$

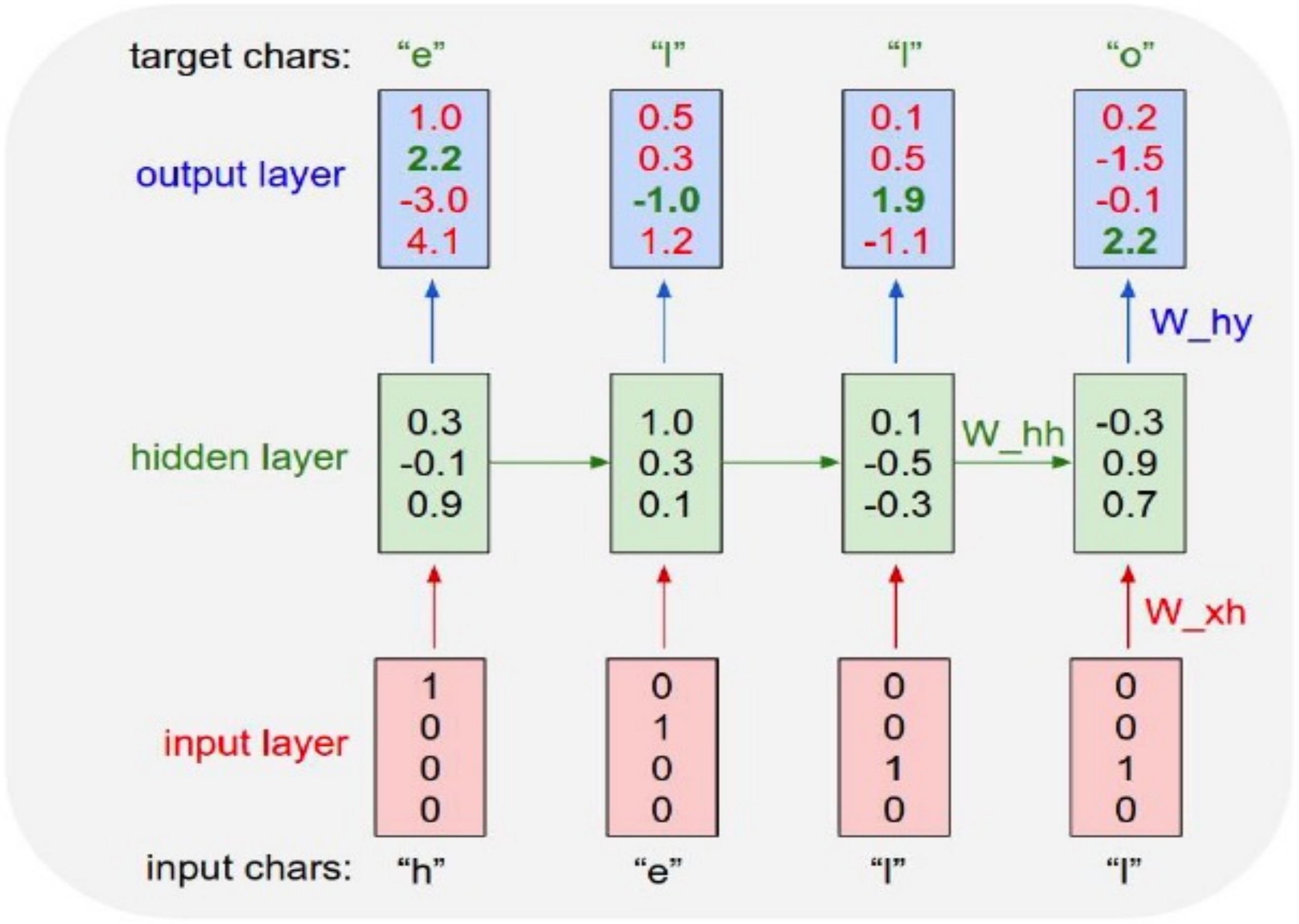


RNN: Example Training

Example: Character-level Language Model

Vocabulary: [h,e,l,o]

Example training sequence: **“hello”**

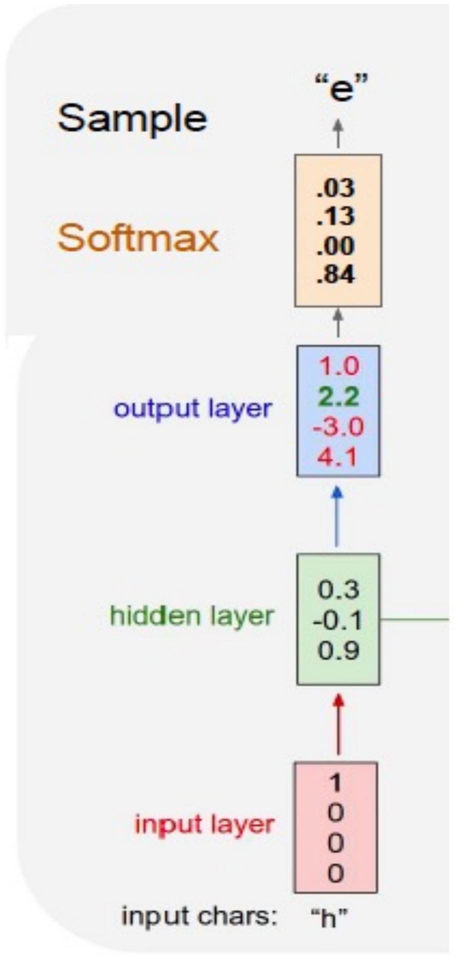


RNN: Example Test

Example:
Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model

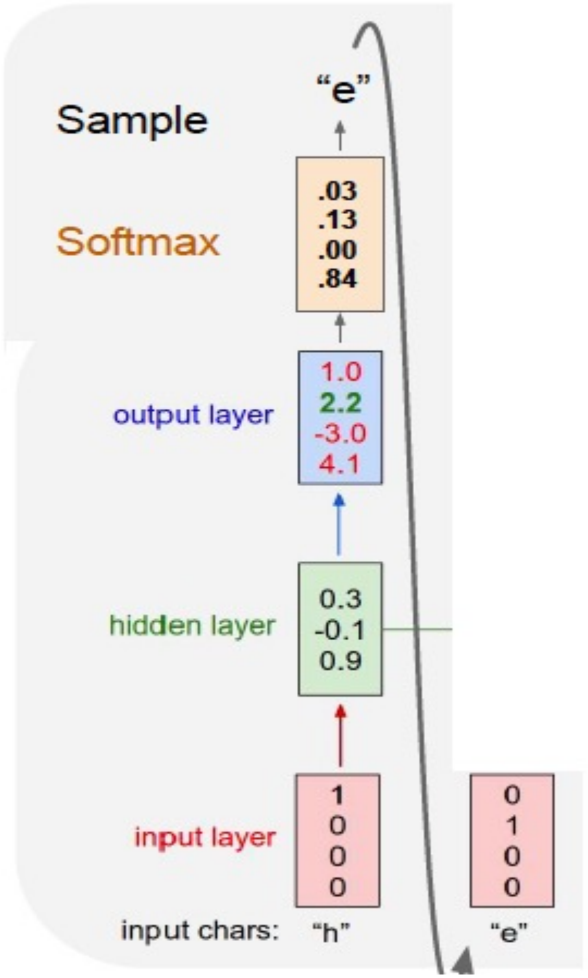


RNN: Example Test

Example: Character-level Language Model Sampling

Vocabulary: [h,e,l,o]

At test-time sample characters one at a time, feed back to model

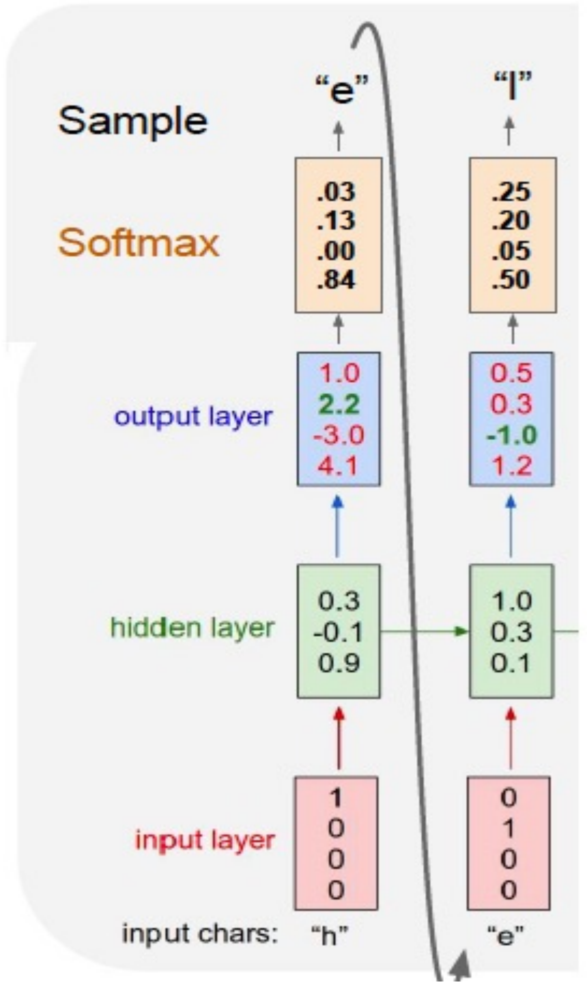


RNN: Example Test

Example: Character-level Language Model Sampling

Vocabulary: [h,e,l,o]

At test-time sample characters one at a time, feed back to model

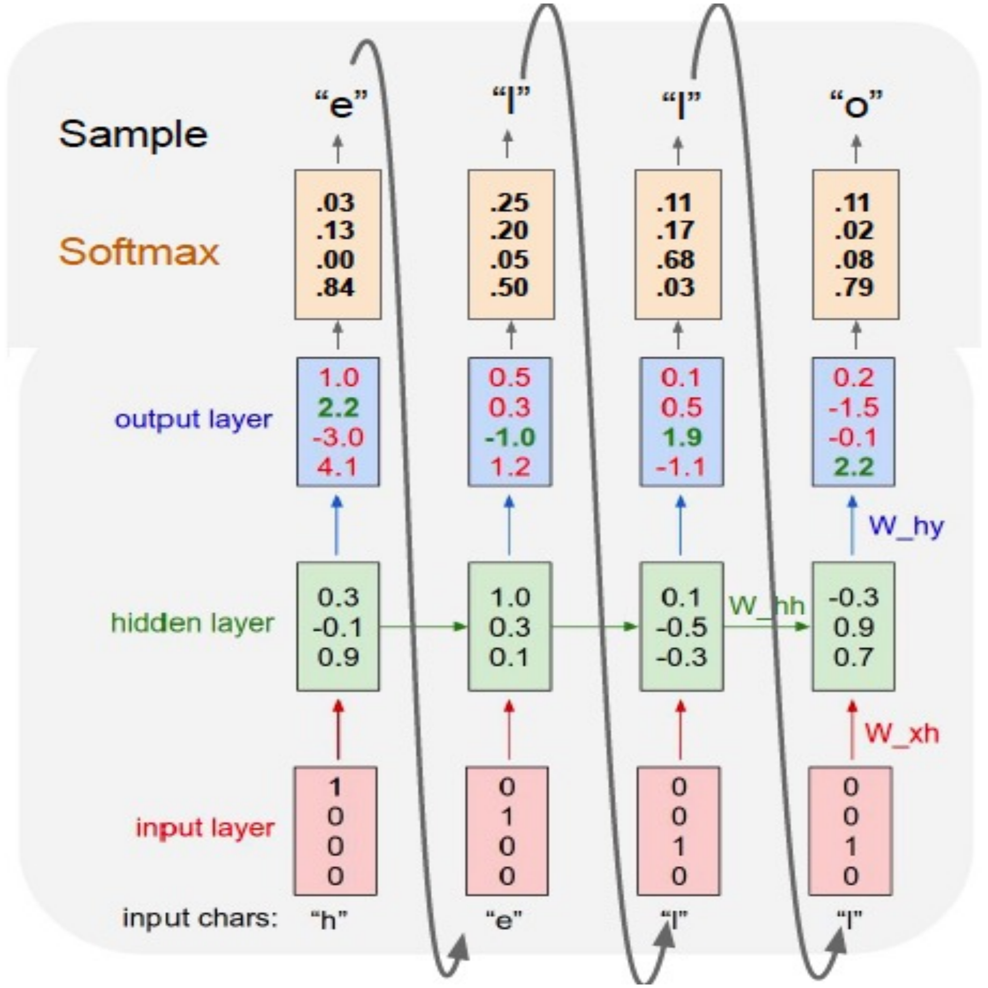


RNN: Example Test

Example: Character-level Language Model Sampling

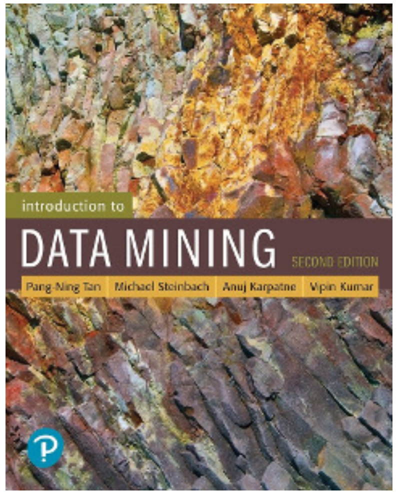
Vocabulary: [h,e,l,o]

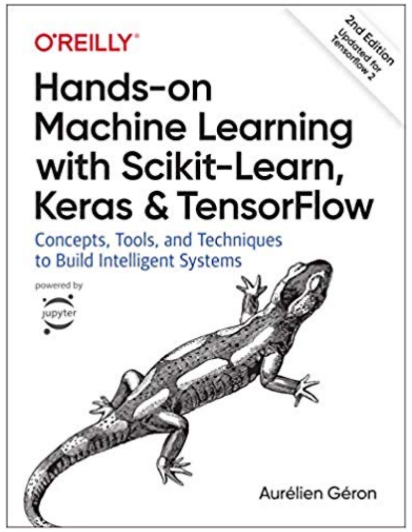
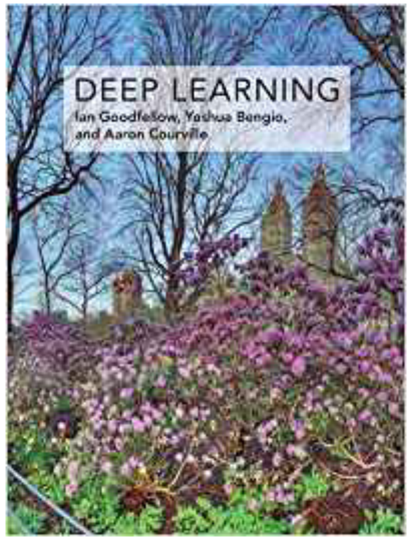
At test-time sample characters one at a time, feed back to model



References

- Artificial Neural Network. Chapter 5.4 and 5.5. Introduction to Data Mining.
- Hands-on Machine Learning with Scikit-Learn, Keras & Tensorflow. A practical handbook to start wrestling with Machine Learning models (2nd ed).
- Deep Learning. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. The reference book for deep learning models.

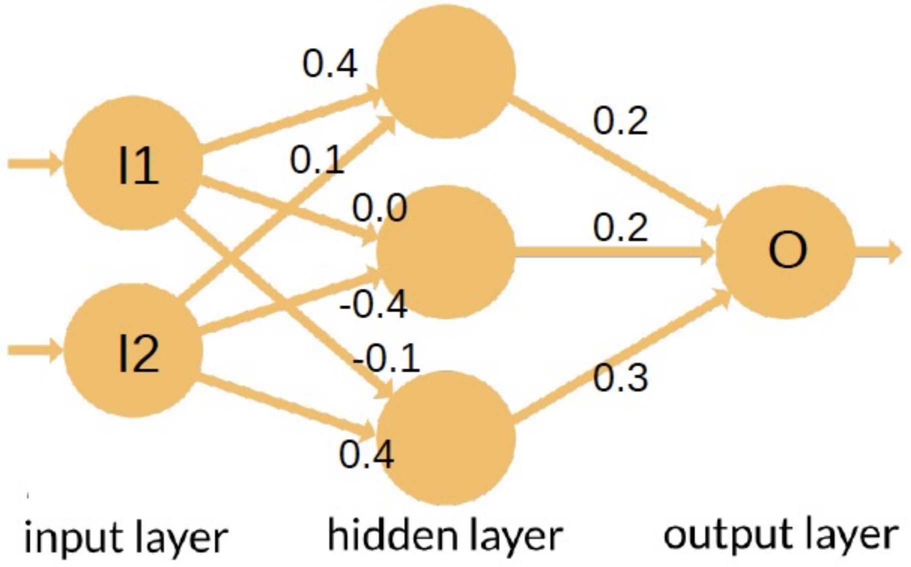



O'REILLY
2nd Edition Updated for TensorFlow 2
Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow
Concepts, Tools, and Techniques to Build Intelligent Systems
powered by jupyter
Aurélien Géron


Exercises - Neural Network

Predict with a Neural Network

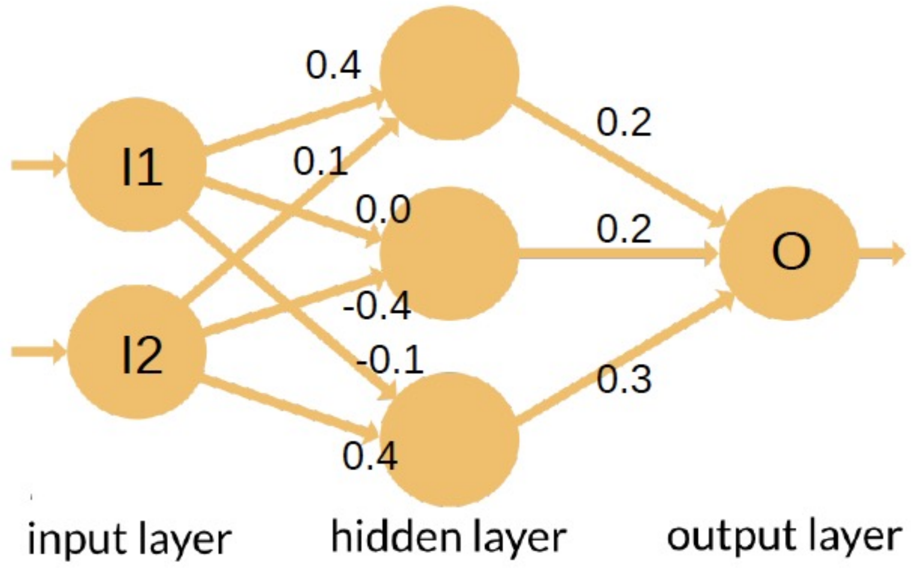
- Given the following NN with
  - assigned weights (see figure)
  - activation function f(S) = sign(S-0.2) for all nodes
- Label the test set on the right, then compute accuracy, and precision & recall for both classes



I1	I2	O
-1	+1	
+1	+1	
+1	-1	
+1	-1	
-1	+1	
+1	+1	
-1	-1	
+1	+1	
-1	-1	
+1	+1	

Predict with a Neural Network - Solution

- Given the following NN with
  - assigned weights (see figure)
  - activation function f(S) = sign(S-0.2) for all nodes
- Label the test set on the right, then compute accurac
  and precision & recall for both classes



I1	I2	C
-1	+1	-1
+1	+1	+1
+1	-1	-1
+1	-1	+1
-1	+1	+1
+1	+1	+1
-1	-1	-1
+1	+1	-1
-1	-1	-1
+1	+1	+1

H_1 = sign(0.4 * -1 + 0.1 * 1 -0.2) =
= sign(-0.5) = -1

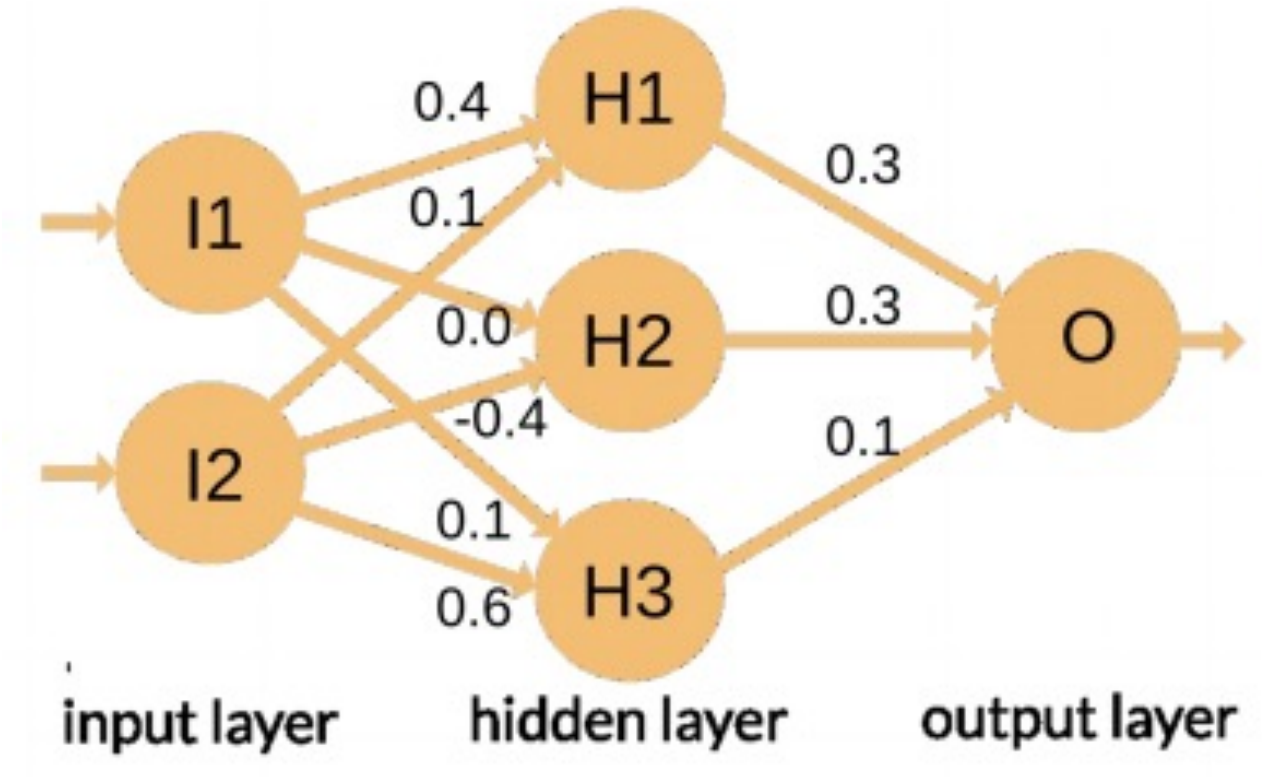
H_2 = sign(0.0 * -1 + -0.4 * 1 -0.2) =
= sign(-0.6) = -1

H_3 = sign(-0.1 * -1 + 0.4 * 1 -0.2) =
= sign(0.3) = 1

Y_1 = sign(0.2 * -1 + 0.2* -1 + 0.3 * 1 -0.2) =
= sign(-0.3) = -1

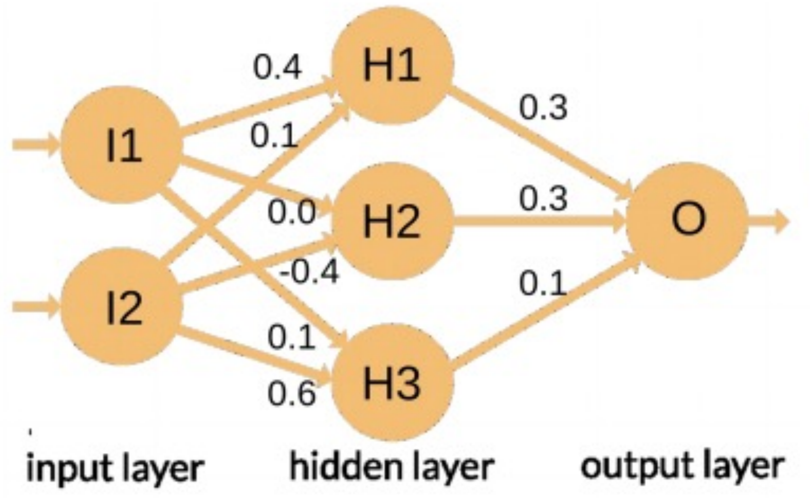
Predict with a Neural Network

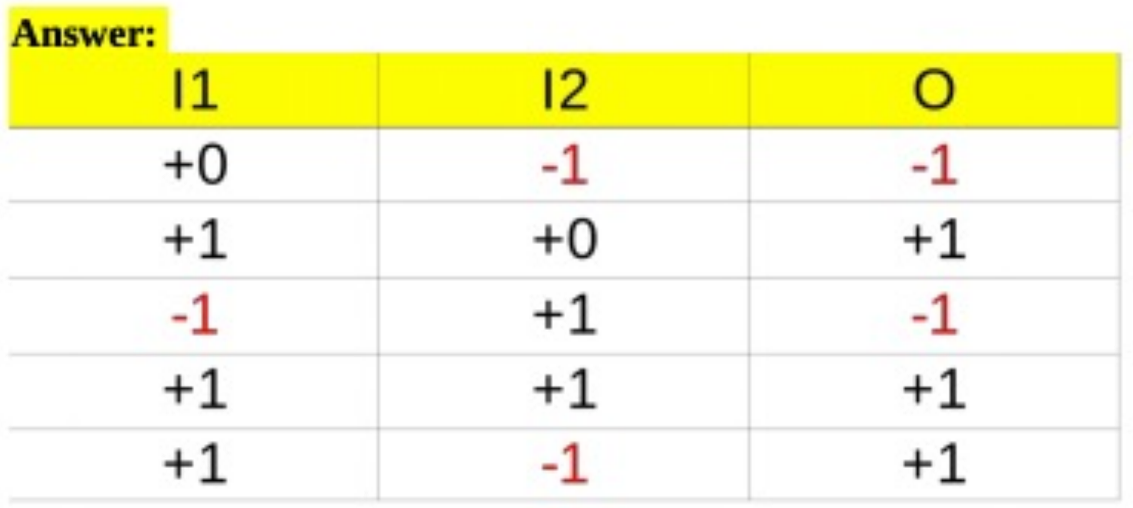
Given the neural network below (on the left), apply it to the test set provided (on the right). The weights are reported beside each connection, while the activation function is simply f(S) = sign(S), i.e. -1 for positive values, +1 for positive ones and 0 for S=0. For each case, show the output also of the nodes on the hidden layer.



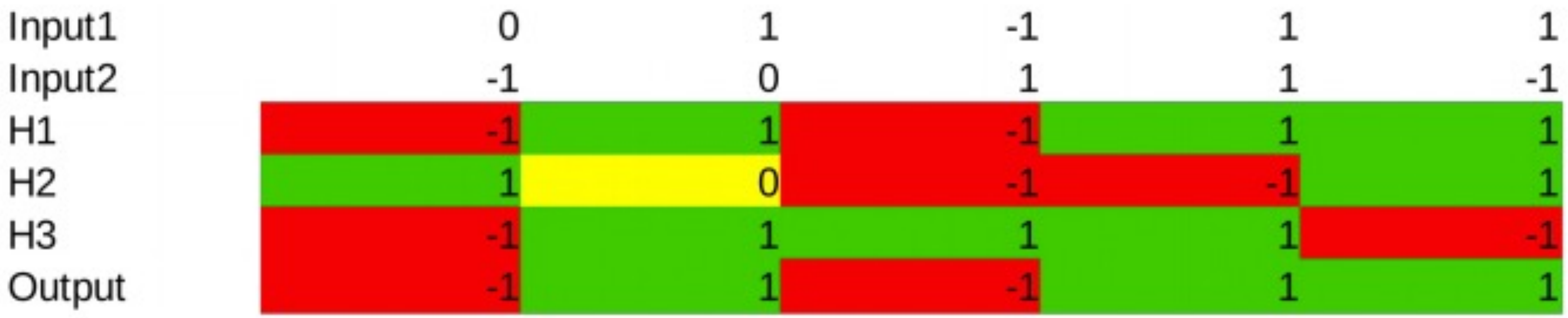
I1	I2	O
+0	-1	
+1	+0	
-1	+1	
+1	+1	
+1	-1	

Predict with a Neural Network - Solution




Answer:

I1	I2	O
+0	-1	-1
+1	+0	+1
-1	+1	-1
+1	+1	+1
+1	-1	+1



Input1	0	1	-1	1	1
Input2	-1	0	1	1	-1
H1	-1	1	-1	1	1
H2	1	0	-1	-1	1
H3	-1	1	1	1	-1
Output	-1	1	-1	1	1