Feed Forward Neural Networks

TextBook:

1.Deep Learning with Python: A Hands-on Introduction Nikhil Ketkar [chapter3]

2. https://towardsdatascience.com/notes-on-artificial-intelligence-ai-machine-learning-ml-anddeep-learning-dl-for-56e51a2071c2

1.Deep Learning with Python: A Hands-on Introduction Nikhil Ketkar

- Chapter3[Unit,Expressing Neural network in Vector form,Output evaluation,Types of units / Activation functions /Layers]
- Chapter5[CNN]
- Chapter6[RNN]

2. https://towardsdatascience.com/notes-on-artificial-intelligence-ai-machine-learning-ml-anddeep-learning-dl-for-56e51a2071c2

- Generative Adversarial Networks
- Natural Language Processing

CONTENTS

- Unit
- Expressing Neural network in Vector form
- Output evaluation
- Types of units / Activation functions /Layers
- Deep Learning: Convolutional neural networks (CNN)
- Recursive (Recurrent) Neural Networks (RNN),
- Generative Adversarial Networks,
- Natural Language Processing

Connectionist Models

Consider humans:

- Neuron switching time ~ .001 second
- Number of neurons ~ 10^{10}
- Connections per neuron ~ 10^{4-5}
- Scene recognition time ~ .1 second
- 100 inference steps doesn't seem like enough

$\rightarrow$ much parallel computation

Properties of artificial neural nets (ANN's):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

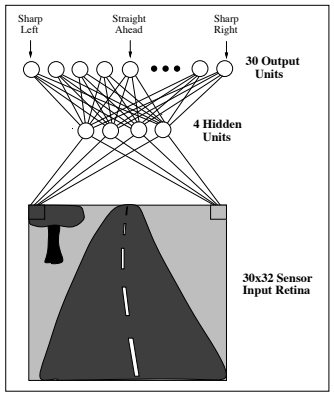
When to Consider Neural Networks

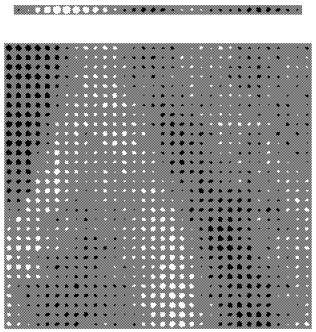
- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

Examples:

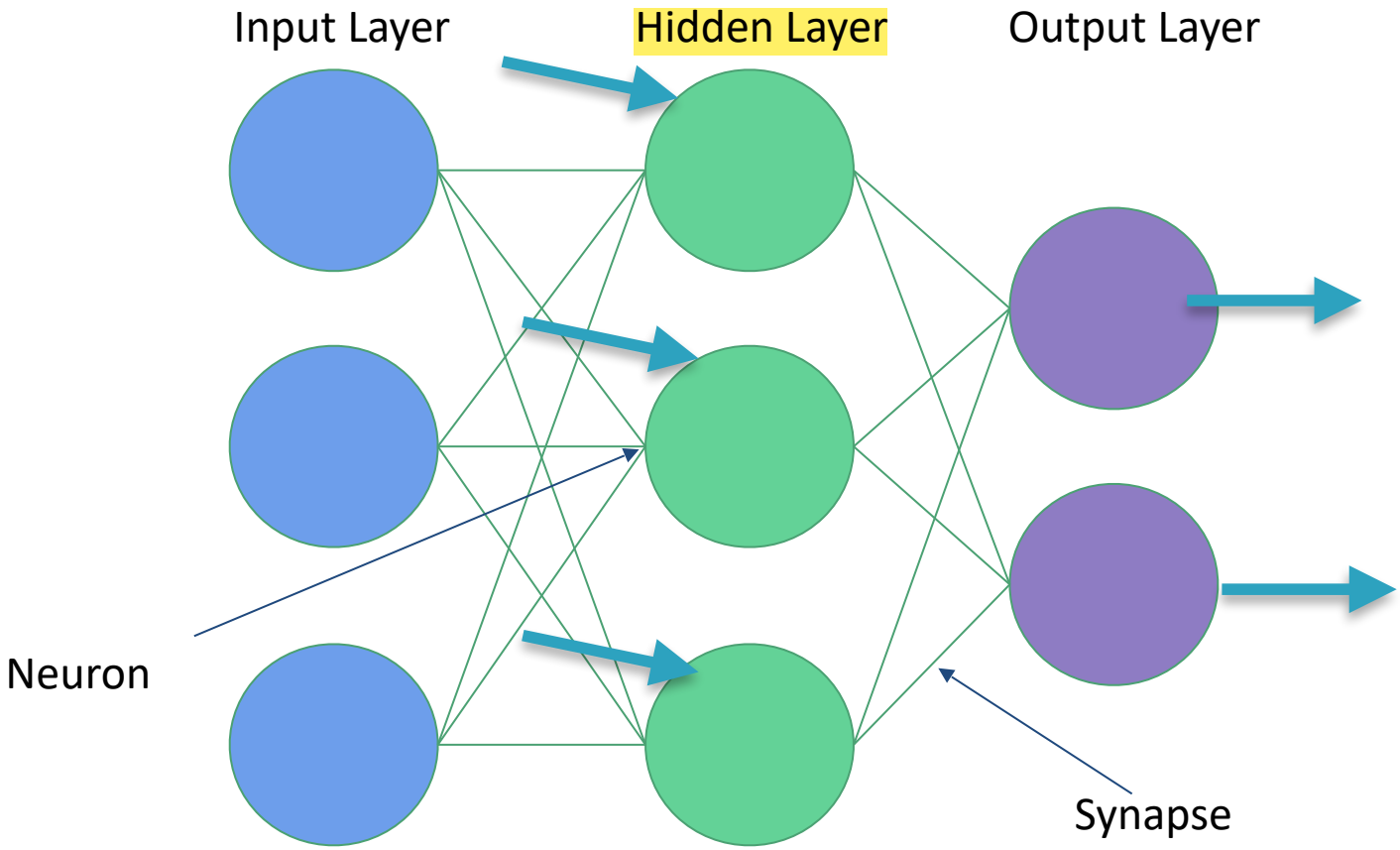
- Speech phoneme recognition [Waibel]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction

ALVINN drives 70 mph on highways

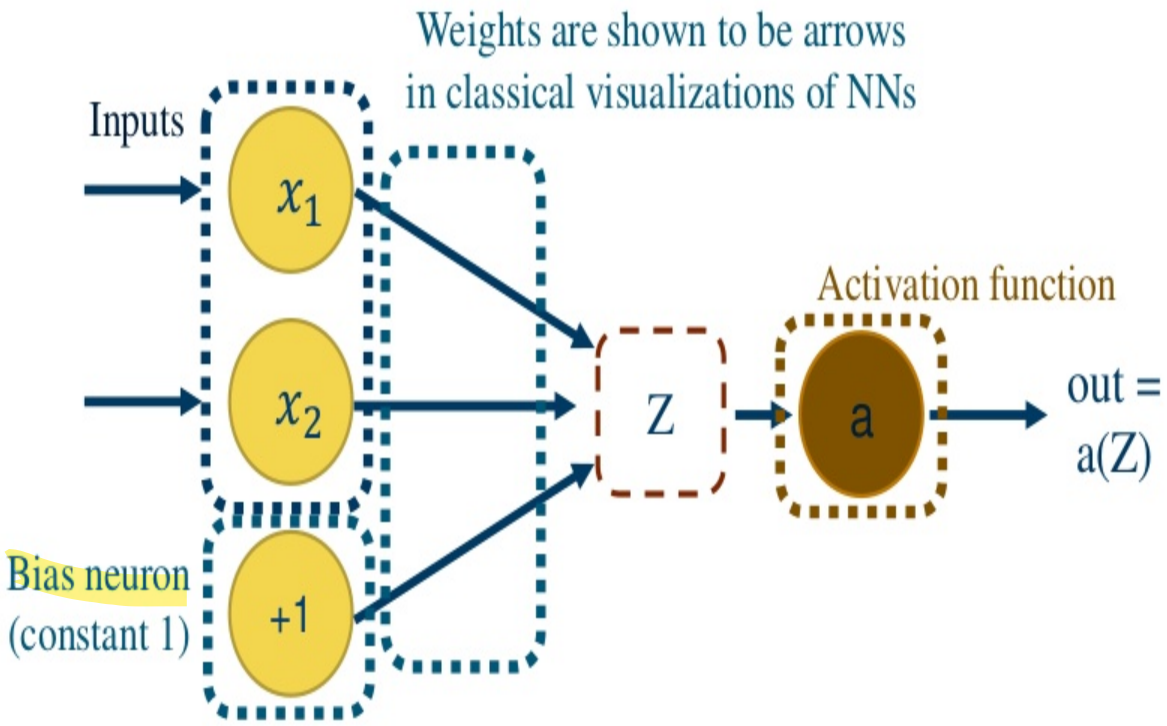




Neural Network Architecture



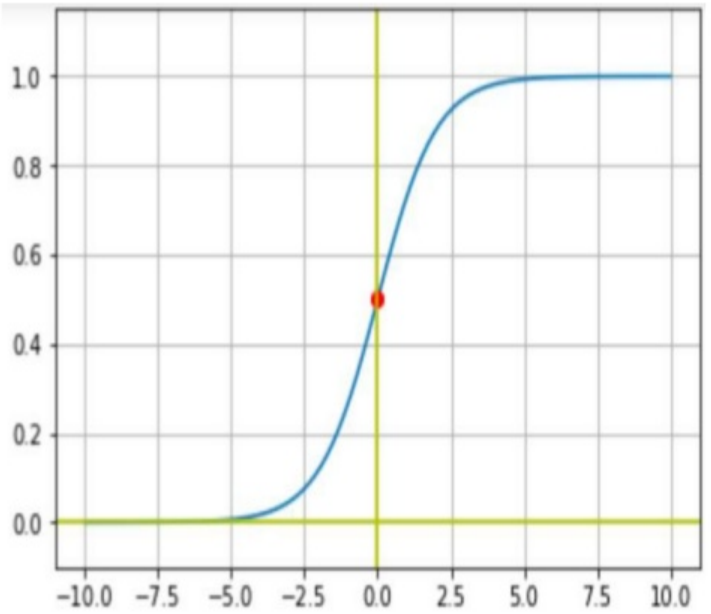
Classic Visualization of Neurons



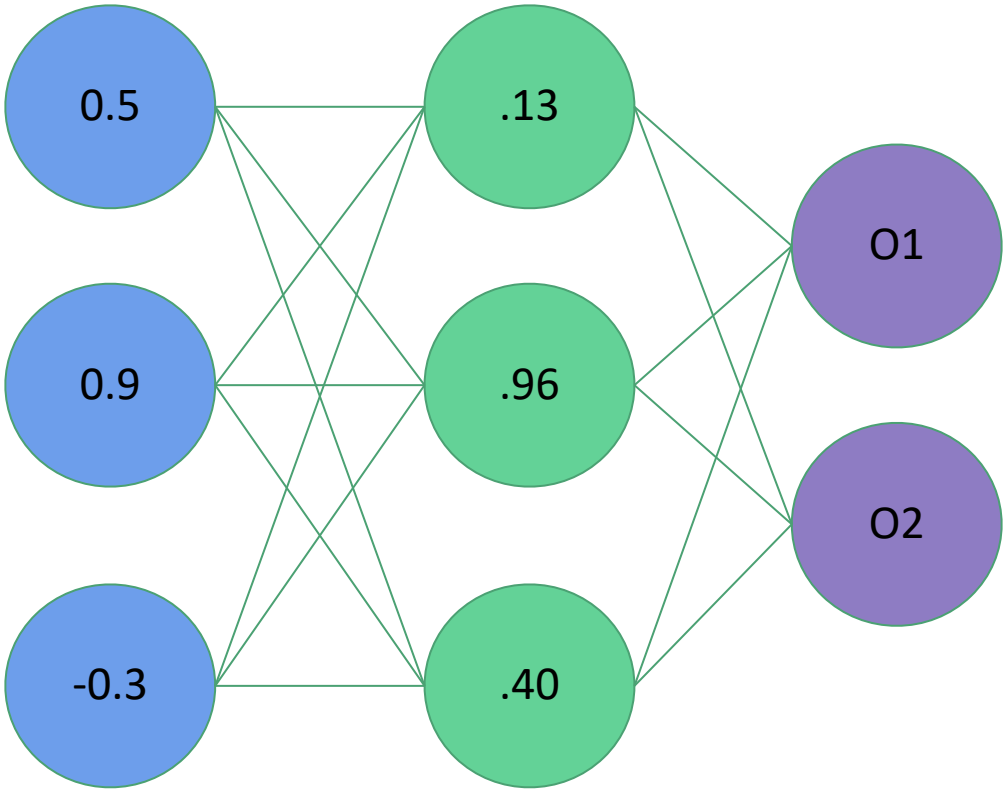
Activation Function: Sigmoid

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

- Value at $z \ll 0$? ≈ 0
- Value at $z = 0$? $= 0.5$
- Value at $z \gg 0$? ≈ 1



Inference

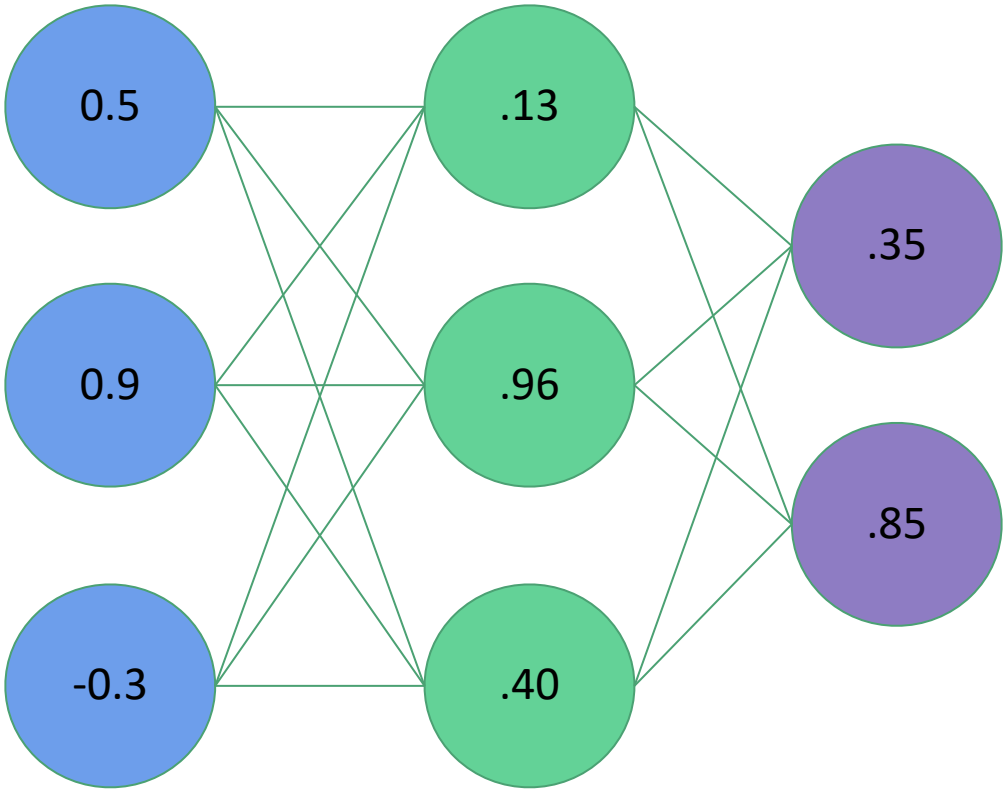


H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

H1 = S(0.5 * 1.0 + 0.9 * -2.0 + -0.3 * 2.0) = S(-1.9) = .13
H2 = S(0.5 * 2.0 + 0.9 * 1.0 + -0.3 * -4.0) = S(3.1) = .96
H3 = S(0.5 * 1.0 + 0.9 * -1.0 + -0.3 * 0.0) = S(-0.4) = .40

Inference



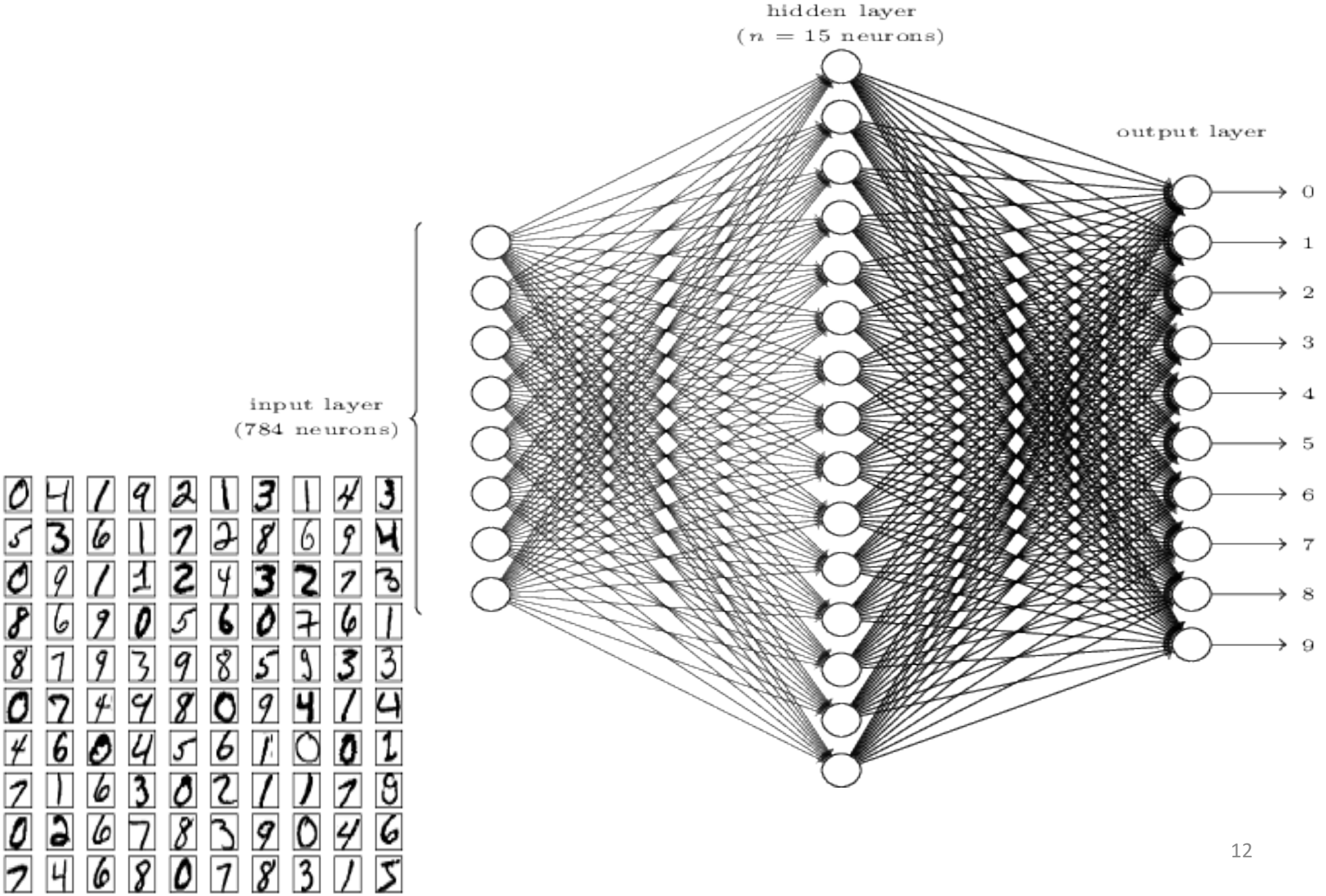
H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

O1 = S(.13 * -3.0 + .96 * 1.0 + .40 * -3.0) = S(-.63) = .35

O1 = S(.13 * 0.0 + .96 * 1.0 + .40 * 2.0) = S(1.76) = .85

Digit Recognition - Neural Nets



UNIT

abstract level, a Neural Network can be thought of as a function $f_\theta : x \to y$, which takes an input $x \in \mathbb{R}^n$ and produces an output $y \in \mathbb{R}^m$, and whose behavior is parameterized by $\theta \in \mathbb{R}^p$. So for instance, f_θ could be simply $y = f_\theta(x) = \theta \cdot x$.

A unit is the basic building of a neural network; refer to Figure 3-1. The following points should be noted

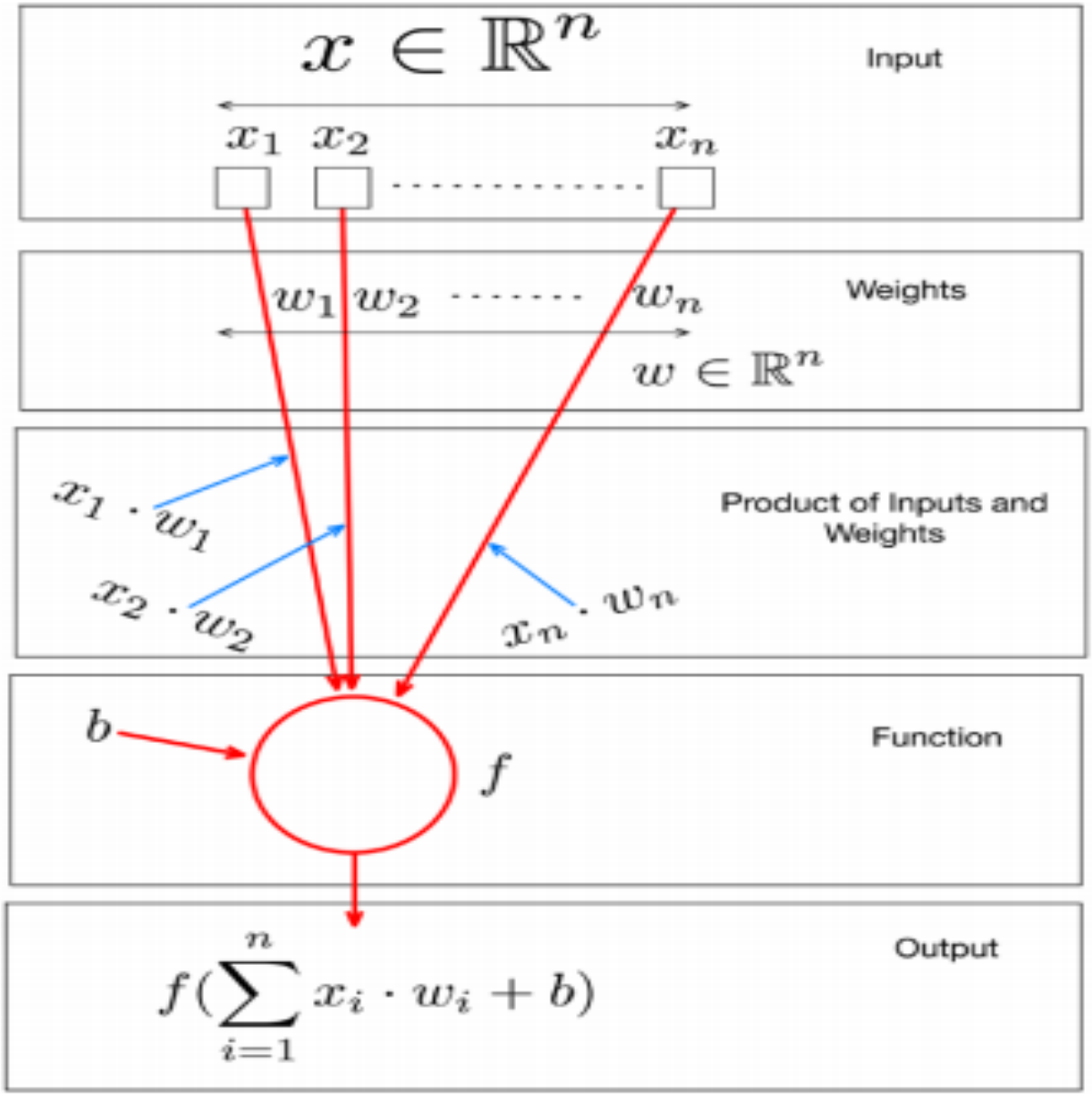
1. A unit is a function that takes as input a vector $x \in \mathbb{R}^n$ and produces a scalar.
2. A unit is parameterized by a weight vector $w \in \mathbb{R}^n$ and a bias term denoted by b.
3. The output of the unit can be described as

$$f\left(\sum_{i=1}^{n} x_i \cdot w_i + b\right)$$

where $f : \mathbb{R} \to \mathbb{R}$ is referred to as an activation function.

4. A variety of activation functions may be used, as we shall see later in the chapter; generally, it's a non-linear function.

UNIT


$x \in \mathbb{R}^n$
Input
x_1 x_2 x_n
w_1 w_2 w_n
Weights
$w \in \mathbb{R}^n$
$x_1 \cdot w_1$
$x_2 \cdot w_2$
$x_n \cdot w_n$
Product of Inputs and Weights
b
f
Function
$f(\sum_{i=1}^{n} x_i \cdot w_i + b)$
Output

UNIT

Neural Networks are constructed using the unit as a basic building block (introduced earlier); refer to Figure

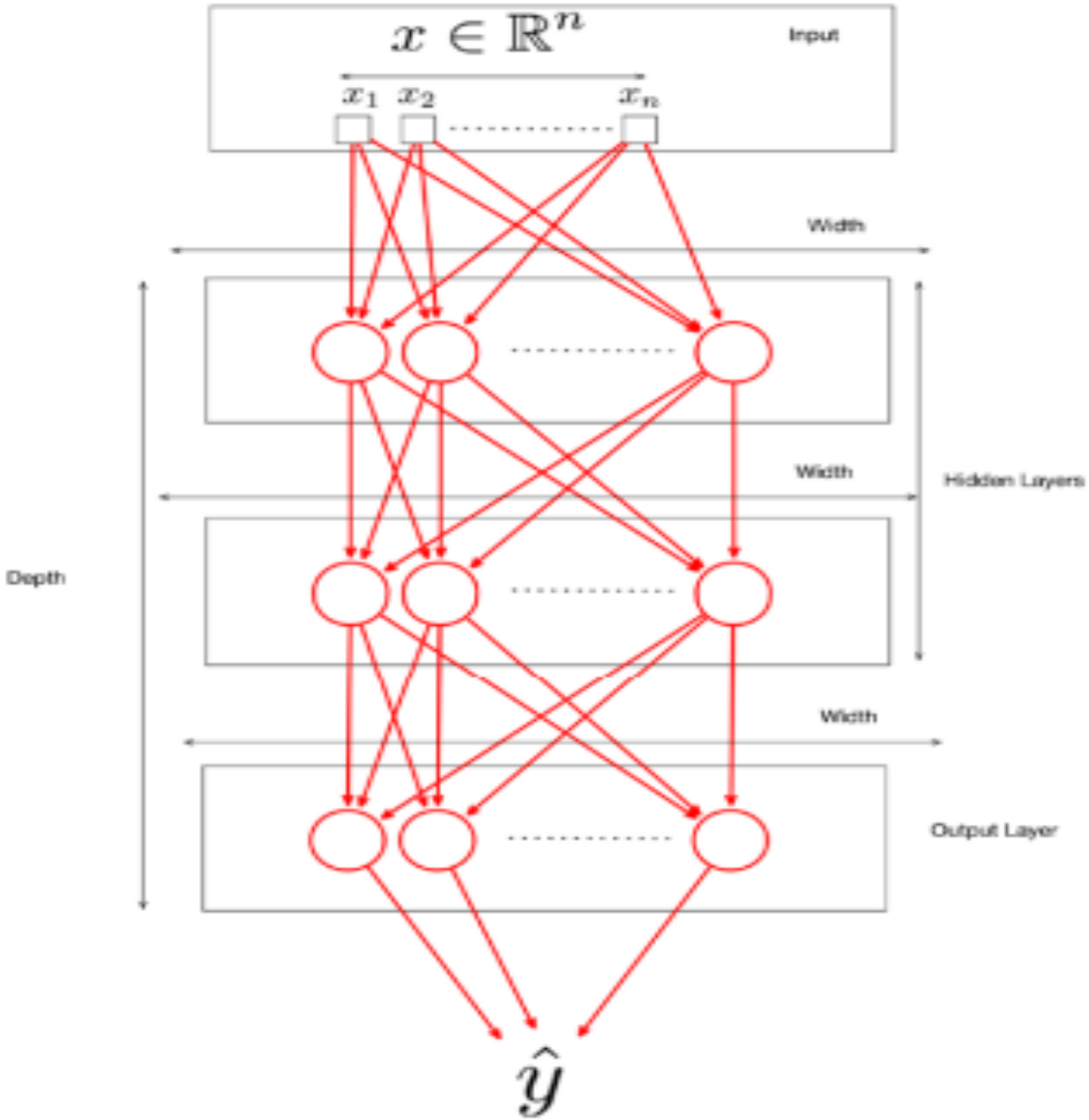
1. Units are organized as **layers**, with every layer containing one or more units.
2. The **last layer** is referred to as the **output layer**.
3. All layers before the output layers are referred to as **hidden layers**.
4. The **number of units in a layer** is referred to as the **width** of the layer
5. The width of each layer need not be the same, but the dimension should be aligned.

UNIT

6. The **number of layers** is referred to as the **depth** of the network. This is where the notion of deep (as in deep learning) comes from.
7. Every layer takes as input the output produced by the previous layer, except for the first layer, which consumes the input.
8. The output of the last layer is the output of the network and is the

9. As we have seen earlier, a neural network can be seen as a function $f_\theta : x \to y$, which takes as input $x \in \mathbb{R}^n$ and produces as output $y \in \mathbb{R}^m$ and whose behavior is parameterized by $\theta \in \mathbb{R}^p$. We can now be more precise about θ; it's simply a collection of all the weights w for all the units in the network.

10. Designing a neural network involves, amongst other things, defining the overall structure of the network, including the number of layers and the width of these layers.


$x \in \mathbb{R}^n$
Input
x_1 x_2 x_n
Width
Width
Hiddlan Layers
Depth
Width
Output Layer
$\hat{y}$

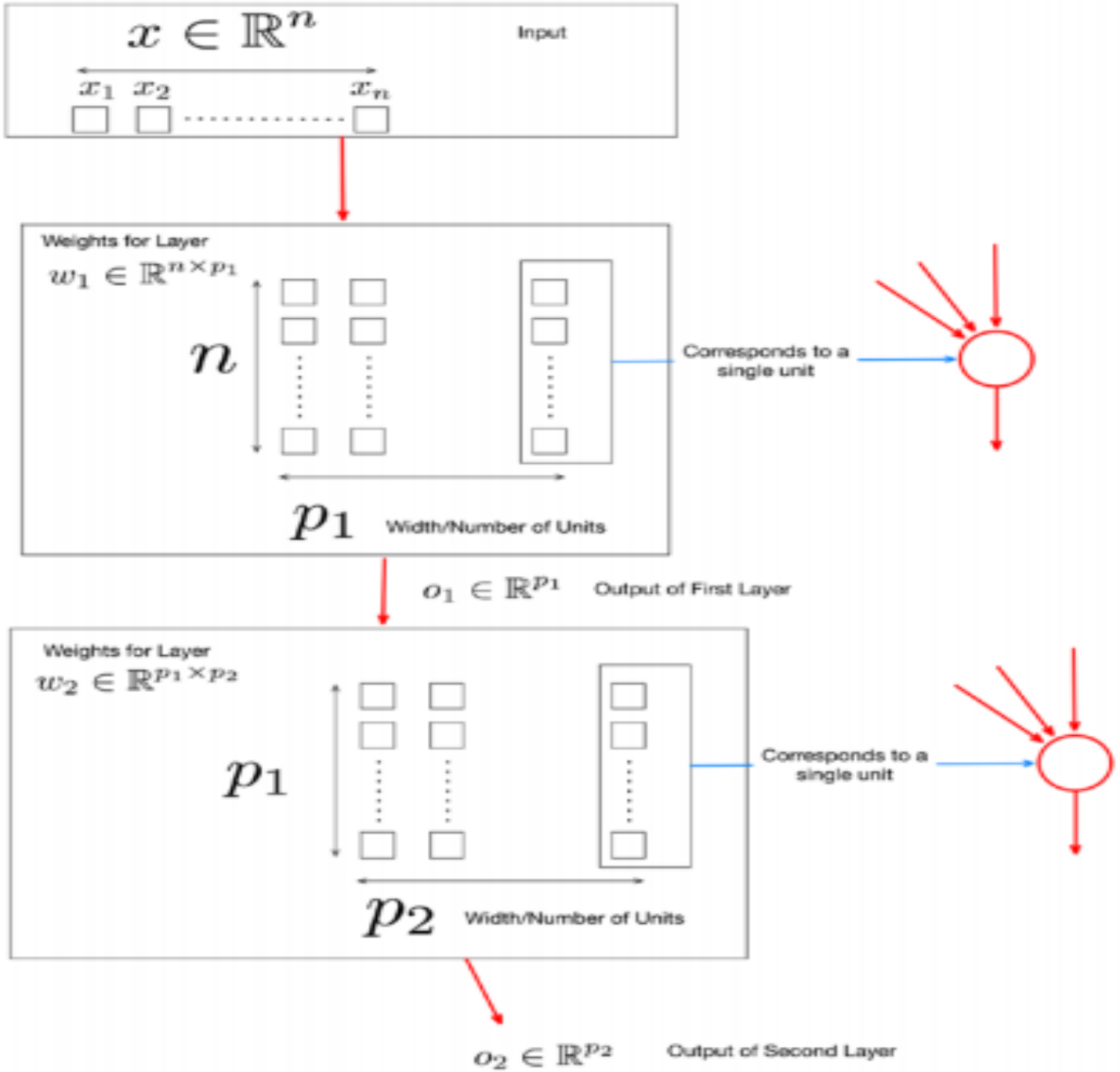
Expressing Neural network in Vector form

1. If we assume that the dimensionality of the input is $x \in \mathbb{R}^n$ and the first layer has p_1 units, then each of the units has $w \in \mathbb{R}^n$ weights associated with them. That is, the weights associated with the first layer are a matrix of the form $w_1 \in \mathbb{R}^{n \times p_1}$. While this is not shown in the diagram, each of the p_1 units also has a bias term associated with it.

2. The first layer produces an output $o_1 \in \mathbb{R}^{p_1}$ where $o_i = f\left(\sum_{k=1}^{n} x_k \cdot w_k + b_i\right)$. Note that the index k corresponds to each of the inputs/weights (going from $1 \ldots n$) and the index i corresponds to the units in the first layer (going from $1..p_1$).

3. Let us now look at the output of the first layer in a vectorised notation. By vectorised notation we simply mean linear algebraic operations like vector matrix multiplications and computation of the activation function on a vector producing a vector (rather than scalar to scalar). The output of the first layer can be represented as $f(x \cdot w_1 + b_1)$. Here we are treating the input $x \in \mathbb{R}^n$ to be of dimensionality $1 \times n$, the weight matrix w_1 to be of dimensionality $n \times p_1$, and the bias term to be a vector of dimensionality $1 \times p_1$. Notice then that $x \cdot w_1 + b$ produces a vector of dimensionality $1 \times p_1$ and the function f simply transforms each element of the vector to produce $o_1 \in \mathbb{R}^{p_1}$.

4. A similar process follows for the second layer that goes from $o_1 \in \mathbb{R}^{p_1}$ to $o_2 \in \mathbb{R}^{p_2}$. This can be written in vectorised form as $f(o_1 \cdot w_2 + b_2)$. We can also write the entire computation up to layer 2 in vectorised form as $f(f(x \cdot w_1 + b_1) \cdot w_2 + b_2)$.


$x \in \mathbb{R}^n$
Input
x_1 x_2 x_n
Weights for Layer
$w_1 \in \mathbb{R}^{n \times p_1}$
n
p_1
Width/Number of Units
Corresponds to a single unit
$o_1 \in \mathbb{R}^{p_1}$
Output of First Layer
Weights for Layer
$w_2 \in \mathbb{R}^{p_1 \times p_2}$
p_1
p_2
Width/Number of Units
Corresponds to a single unit
$o_2 \in \mathbb{R}^{p_2}$
Output of Second Layer

Evaluating the output of the Neural Network

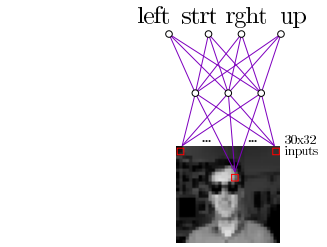
1. We can assume that our data has the form $D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$ where $x \in \mathbb{R}^n$ and $y \in \{0, 1\}$, which is the target of interest (currently this is binary, but it may be categorical or real valued depending on whether we are dealing with a multi-class or regression problem, respectively).
2. For a single data point we can compute the output of the Neural Network, which we denote as $\hat{y}$.
3. Now we need to compute how good the prediction of our Neural Network $\hat{y}$ is as compared to y. Here comes the notion of a loss function.
4. A loss function measures the disagreement between $\hat{y}$ and y which we denote by l. There are a number of loss functions appropriate for the task at hand: binary classification, multi-classification, or regression, which we shall cover later in the chapter (typically derived using Maximum Likelihood).
5. A loss function typically computes the disagreement between $\hat{y}$ and y over a number of data points rather than a single data point.

Face Recognition – The Task

- Classifying faces of people in various poses
- Dataset: 20 people, 32 different images/person
  - in varying expression(happy,sad,angry,neutral)
  - Direction of look(left, right, straight ahead, up)
  - With or without SunGlass
  - Variation in background, clothes of the person etc

  Dataset size: 624 Images
- Image Specification
  - Greyscale (0-255), Resolution 120x128
- Learning Task
  - Recognizing pose
- Achieved accuracy
  - 90% va 25% from random guessing

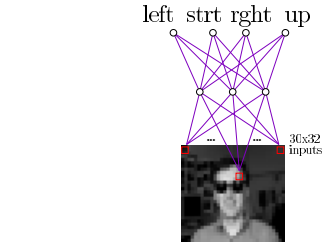
Neural Nets for Face Recognition



Typical input images

90% accurate learning head pose, and recognizing 1-of-20

Exercise



Typical input images

90% accurate learning head pose, and recognizing 1-of-20 faces

- Input encoding
  - 30x32 size normalized images from 120x128
  - Mean of the 16 pixels taken and Intensity scaled between 0 and 1
- Network graph structure. (2 layers)
  - One hidden layer with 3 units (sigmoid activation function),
  - 1 output layer with 4 units (sigmoid activation function)
- Output encoding
  - 1-of-n output encoding one unit for each of the 4 poses
  - Output vectors used in training for the 4 poses

  Left <0.9,0.1,0.1,0.1>, Straight <0.1, 0.9, 0.1,0.1> …..

  Using 0 and 1 will force the weights to grow without bound.

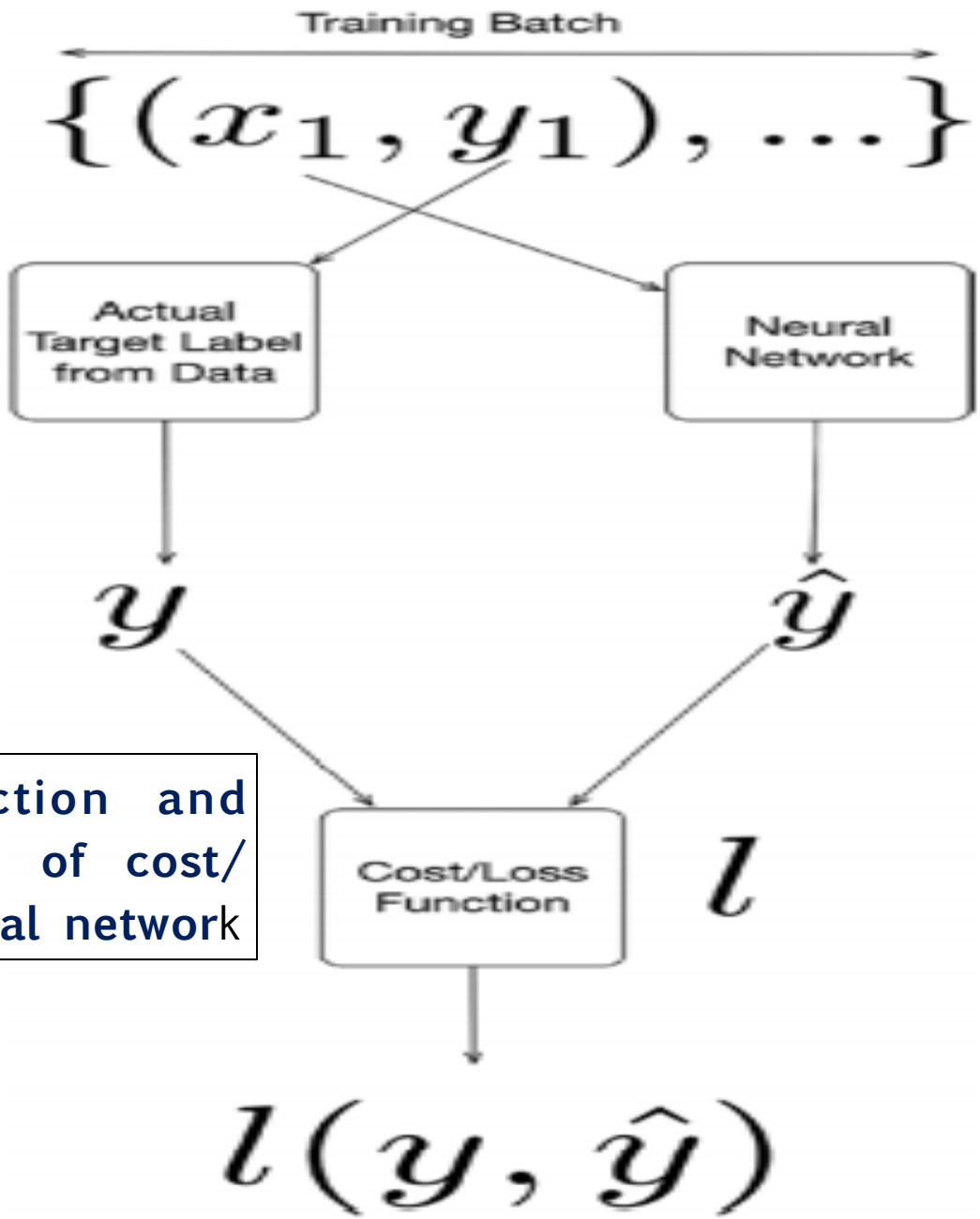
Find the total number of weights in this neural network solution.

Design Choices _ FaceRecognition

- Input encoding
  - 30x32 size normalized images from 120x128
  - Mean of the 16 pixels taken and Intensity scaled between 0 and 1
- Output encoding
  - 1-of-n output encoding one unit for each of the 4 poses
  - Output vectors used in training for the 4 poses

  Left <0.9,0.1,0.1,0.1>, Straight <0.1, 0.9, 0.1,0.1> …..

  Using 0 and 1 will force the weights to grow without bound.
- Network graph structure
  - 2 layers. One hidden layer with 3 units (sigmoid), 1 output layer with 4 units (sigmoid) 90% accuracy
  - In an experiment that used 30 hidden units in hidden layer only 2% extra accuracy achieved
  - What took 1 hr with 30 hidden units took 0.5 minutes for 3 hidden units network on sun Sparc5 workstation
  - Increasing the number of hidden units increases possibility of overfitting
  - Discussed cross validation method should be used to identify the right number of iterations.
- Other learning algorithm parameters
  - Learning rate $\eta = 0.3$, Momentum factor $\alpha = 0.3$
  - Normal gradient descent was used
  - Input unit weights initialized to 0, output unit weights randomly
  - After every 50 iterations of training, performance validated on validation set
  - 90% accuracy on 1/3rd of the data set not used for training



Loss/Cost function and the computation of cost/ loss w.r.t, a neural network

Training the Neural Network

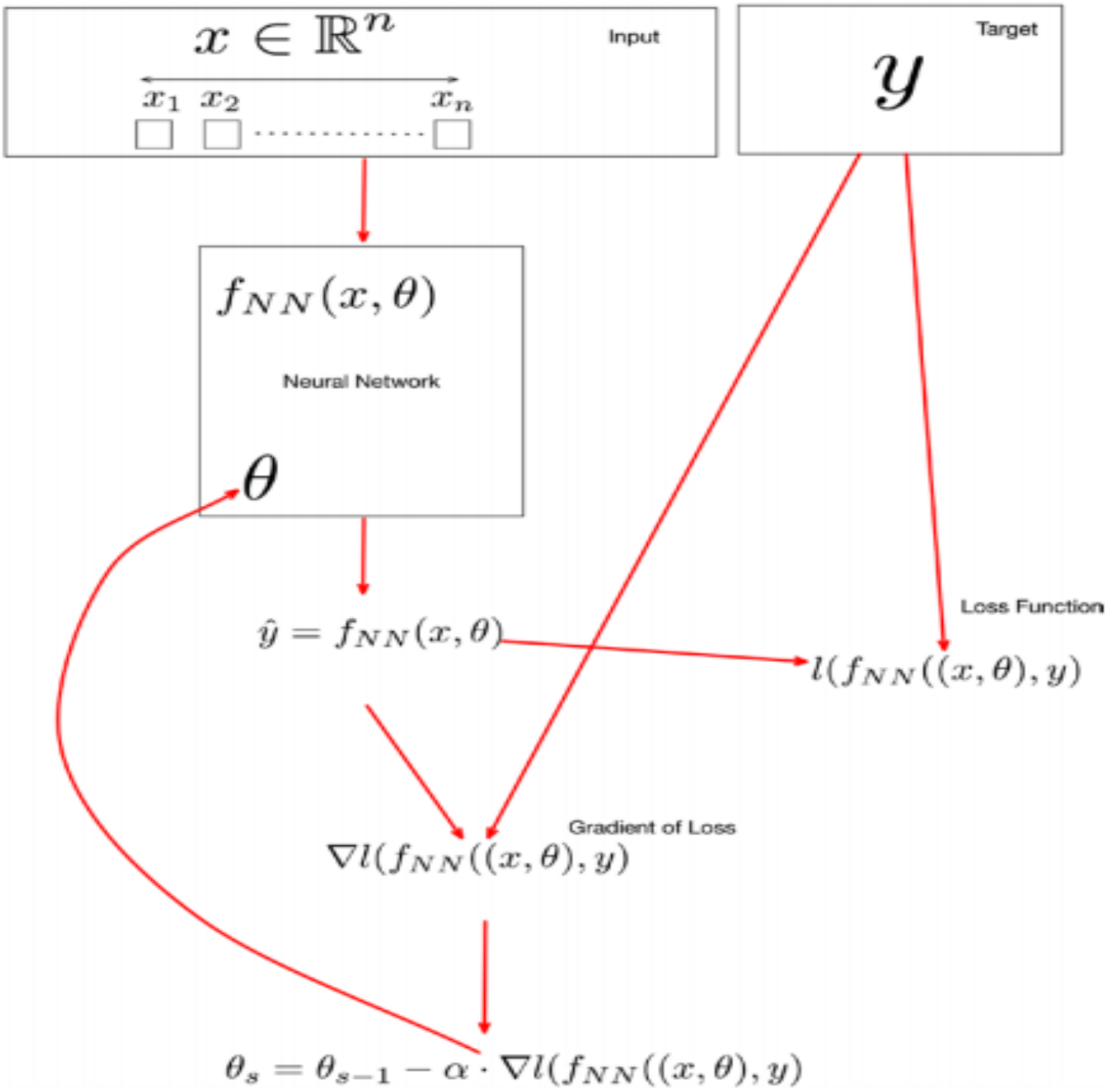
Let us now look at how the Neural Network is Trained. Refer to Figure 3-5. The following points are to be noted:

1. Assuming the same notation as earlier, we denote by θ the collection of all the weights and bias terms of all the layers of the network. Let us assume that θ has been initialized with random values. We denote by f_{NN} the overall function representing the Neural Network.

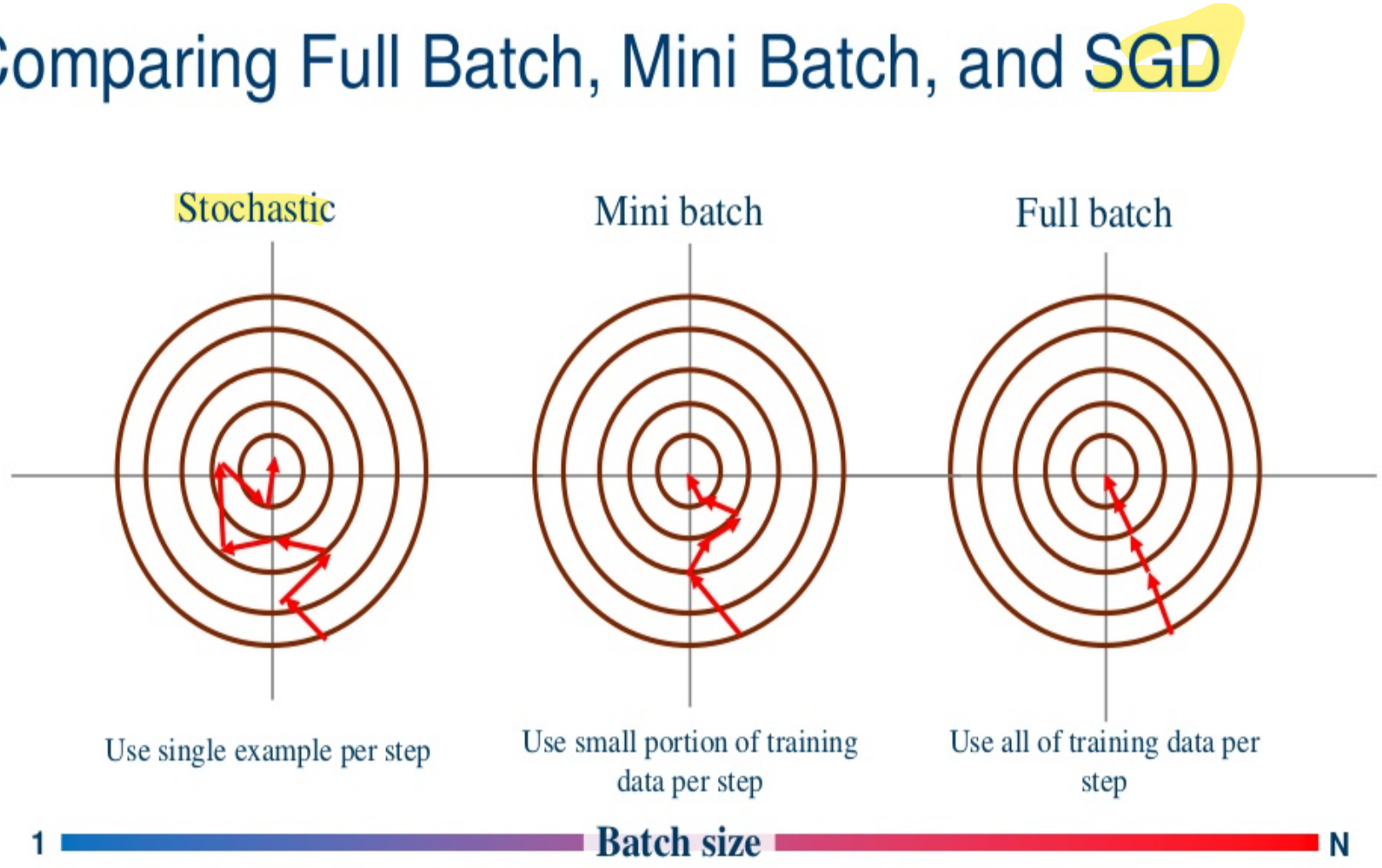
2. As we have seen earlier, we can take a single data point and compute the output of the Neural Network $\hat{y}$. We can also compute the disagreement with the actual output y using the loss function $l(\hat{y}, y)$ that is $l(f_{NN}(x, \theta), y)$.

3. Let us now compute the gradient of this loss function and denote it by $\nabla l\left(f_{NN}(x,\theta), y\right)$.

4. We can now update θ using steepest descent as $\theta_s = \theta_{s-1} - \alpha \cdot l\left(f_{NN}(x,\theta), y\right)$ where s denotes a single step (we can take many such steps over different data points in our training set over and over again until we have a reasonably good value for $l(f_{NN}(x, \theta), y)$.


$x \in \mathbb{R}^n$
Input
x_1 x_2 x_n
Target
y
$f_{NN}(x, \theta)$
Neural Network
θ
$\hat{y} = f_{NN}(x, \theta)$
Loss Function
$l(f_{NN}((x, \theta), y)$
Gradient of Loss
$\nabla l(f_{NN}((x, \theta), y)$
$\theta_s = \theta_{s-1} - \alpha \cdot \nabla l(f_{NN}((x, \theta), y)$

Back propagation and training of weights



Loss Functions

1. The Binary Cross entropy given by the expression

$$-\sum_{i=1}^{n} y_i \log f(x_i,\theta)+(1-y_i)\log(1-f(x_i,\theta))$$

is the recommended loss function for binary classification. This loss function should typically be used when the Neural Network is designed to predict the probability of the outcome. In such cases, the output layer has a single unit with a suitable sigmoid as the activation function.

2. The Cross entropy function given by the expression

$$-\sum_{i=1}^{n} y_i \log f(x_i,\theta)$$

is the recommended loss function for multi-classification. This loss function should typically be used with the Neural Network and is designed to predict the probability of the outcomes of each of the classes. In such cases, the output layer has softmax units (one for each class).

3. The squared loss function given by $\sum_{i=1}^{n}(y-\hat{y})^2$ should be used for regression problems. The output layer in this case will have a single unit.

Types of Units/Activation Functions/ Layers

- In theory, when an activation function is non-linear, a two-layer Neural Network can approximate any function (given a sufficient number of units in the hidden layer). Thus, we do seek non-linear activation functions in general.
- A function that is continuously differentiable allows for gradients to be computed and gradient-based methods to be used for finding the parameters that minimize our loss function over the data. If a function is not continuously differentiable, gradient-based methods cannot make progress.
- A function whose range is finite (as against infinite) leads to a more stable performance w.r.t gradient-based methods.
- Smooth functions are preferred (empirical evidence) and Monolithic functions for a single layer lead to convex error surfaces (this is typically not a consideration w.r.t deep learning). Are symmetric around the origin and behave like identity functions near the origin (f x( ) = x).

Linear Unit

- Linear Unit The Linear unit is the simplest unit which transforms the input as
- $y = w.x + b$
- As the name indicates, the unit does not have a non-linear behavior and is typically used to generate the mean of a conditional Gaussian distribution. Linear units make gradient-based learning a fairly straightforward task.

Sigmoid Unit

The Sigmoid unit transforms the input as follows:

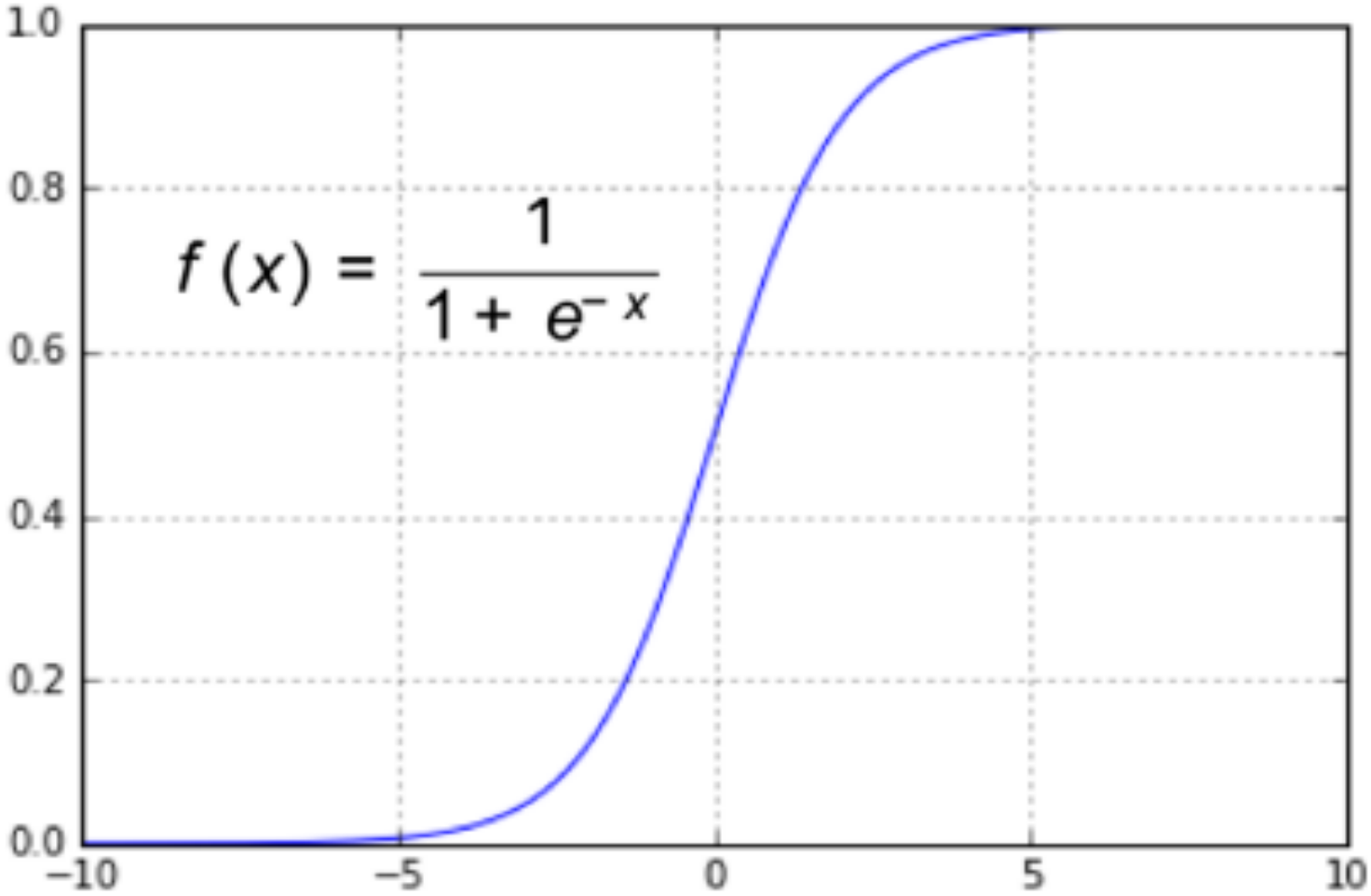
$$y = \frac{1}{1+e^{-(wx+b)}}.$$

The underlying activation function (refer to Figure 3-6) is given by

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

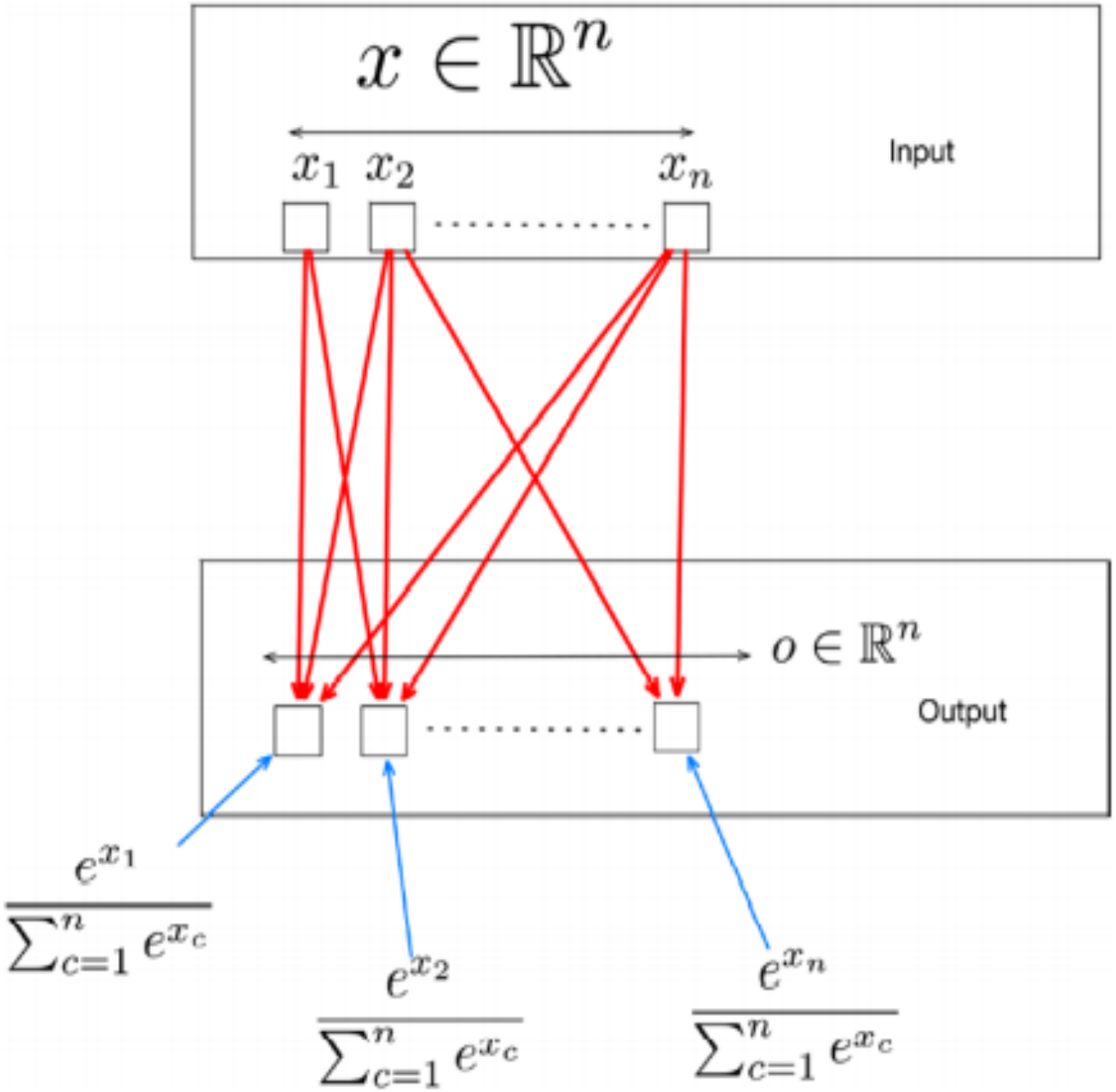
Sigmoid units can be used in the output layer in conjunction with binary cross entropy for binary classification problems. The output of this unit can model a Bernoulli distribution over the output y conditioned over x.

Sigmoid Unit



Softmax Layer

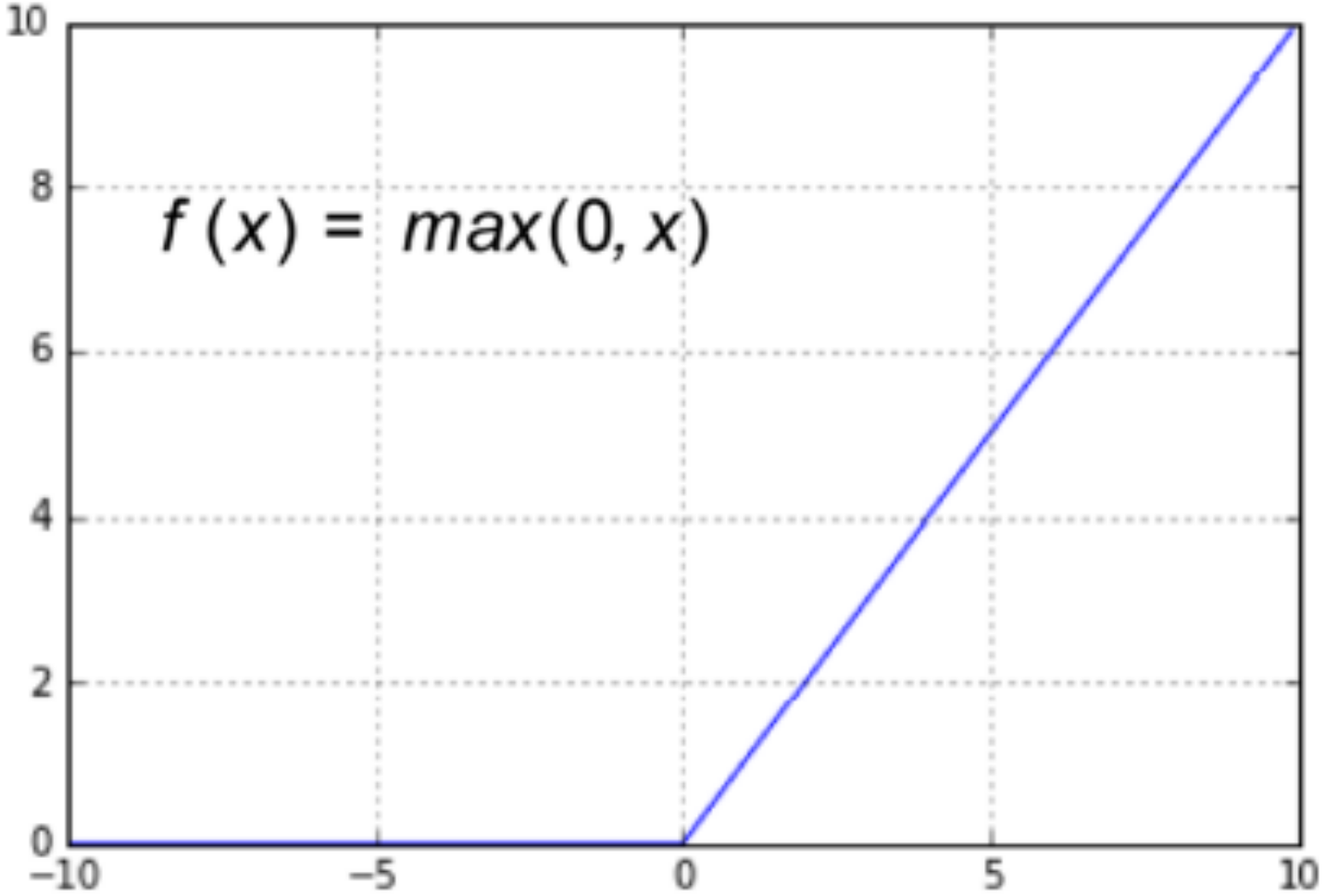
- The Softmax layer is typically used as an output layer for multi-classification tasks in conjunction with the **Cross Entropy loss function**. Refer to Figure 3-7.
- The Softmax layer **normalizes outputs of the previous layer so that they sum up to one**.
- Typically, the units of the previous layer model an un-normalized score of how likely the input is to belong to a particular class. The softmax layer normalized this so that the output represents the probability for every class.


$x \in \mathbb{R}^n$
x_1 x_2 x_n
Input
$o \in \mathbb{R}^n$
Output
$\frac{e^{x_1}}{\sum_{c=1}^{n} e^{x_c}}$
$\frac{e^{x_2}}{\sum_{c=1}^{n} e^{x_c}}$
$\frac{e^{x_n}}{\sum_{c=1}^{n} e^{x_c}}$

Rectified Linear Unit (ReLU)

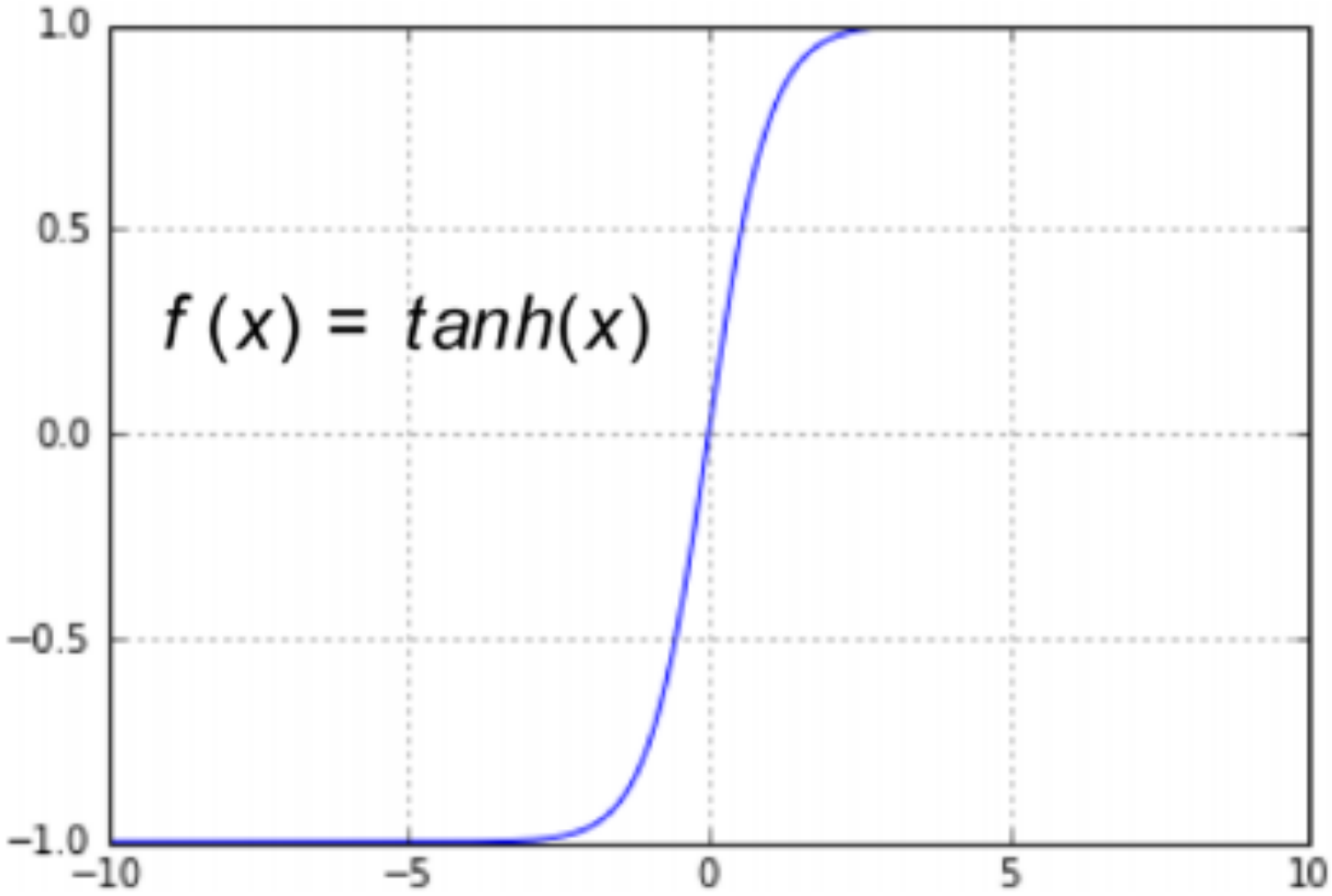
- Rectified Linear Unit used in conjunction with a linear transformation transforms the input as f (x) = max (0, wx+ b).
- The underlying activation function is

  f x( ) = max (0 ,x)
- The ReLU unit is more commonly used as a **hidden unit** in recent times. Results show that ReLU units lead to large and **consistent gradients,** which helps **gradient-based learning**

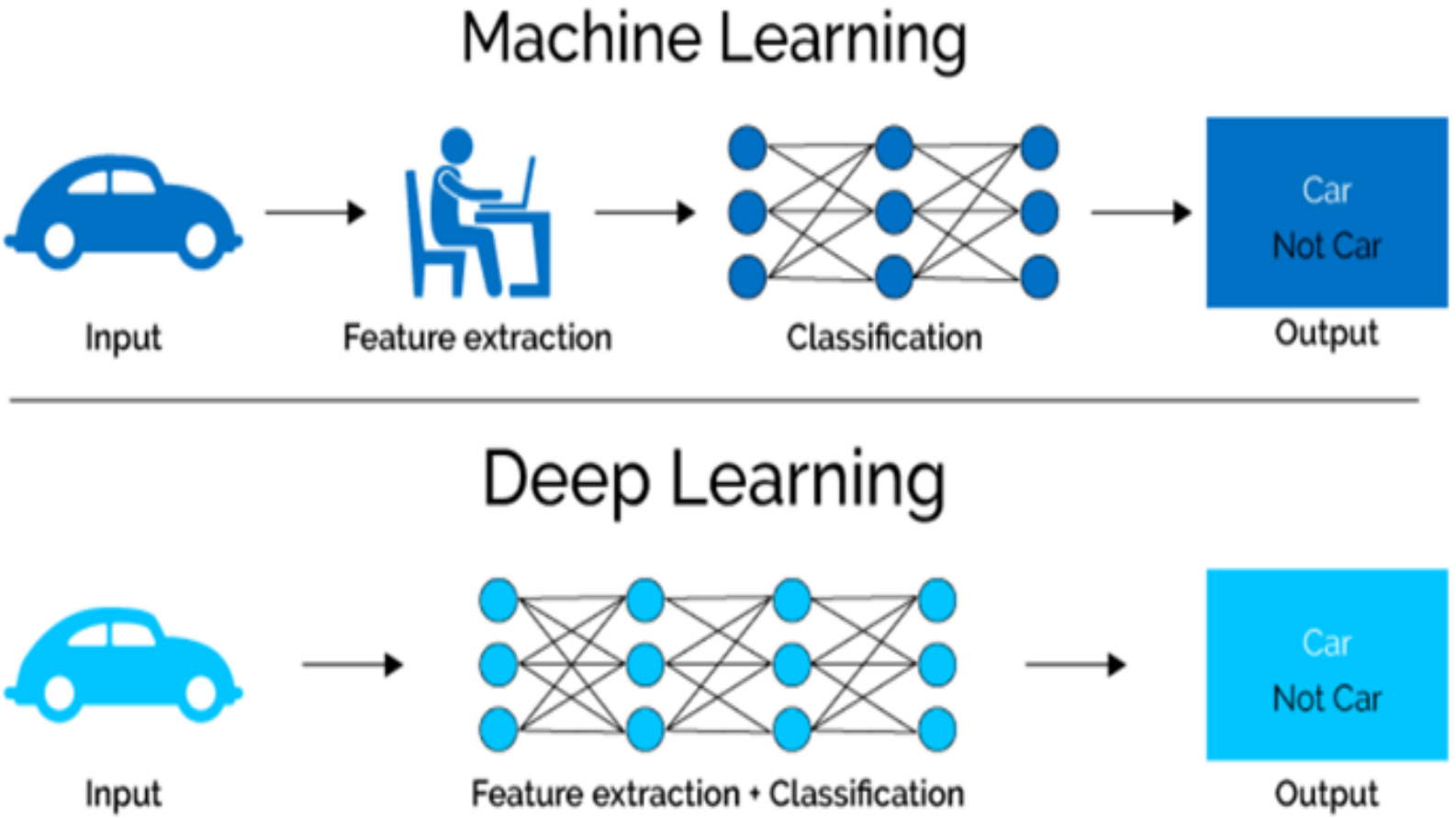

$f(x) = max(0, x)$
10
8
6
4
2
0
−10
−5
0
5
10

Hyperbolic Tangent

- The Hyperbolic Tangent unit transforms the input (used in conjunction with a linear transformation) as follows: $y = \tanh(wx + b)$
- The underlying activation function (refer to Figure 3-9) is given by $f(x) = \tanh(x)$. The hyperbolic tangent unit is also commonly used as a **hidden unit.**


1.0
0.5
0.0
−0.5
−1.0
−10
−5
0
5
10
f (x) = tanh(x)

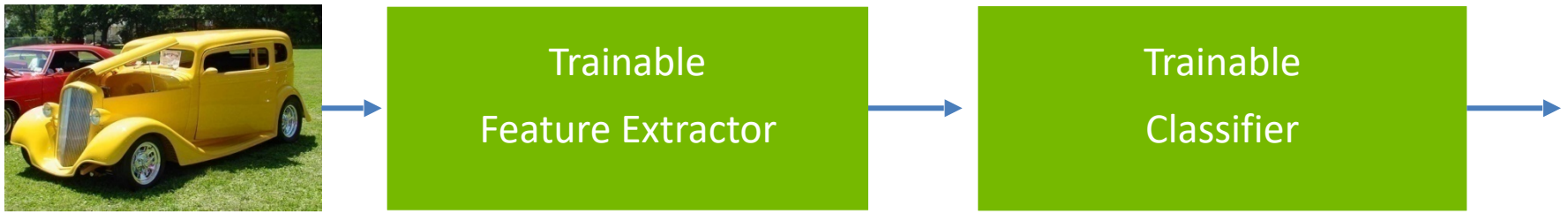
Deep Learning



Deep learning = Learning representations/features and Classification

– The traditional model of pattern recognition (since the late 50's)
  – Fixed/engineered features (or fixed kernel) + trainable classifier

– Deep learning:
  – Trainable features (or kernel) + trainable classifier

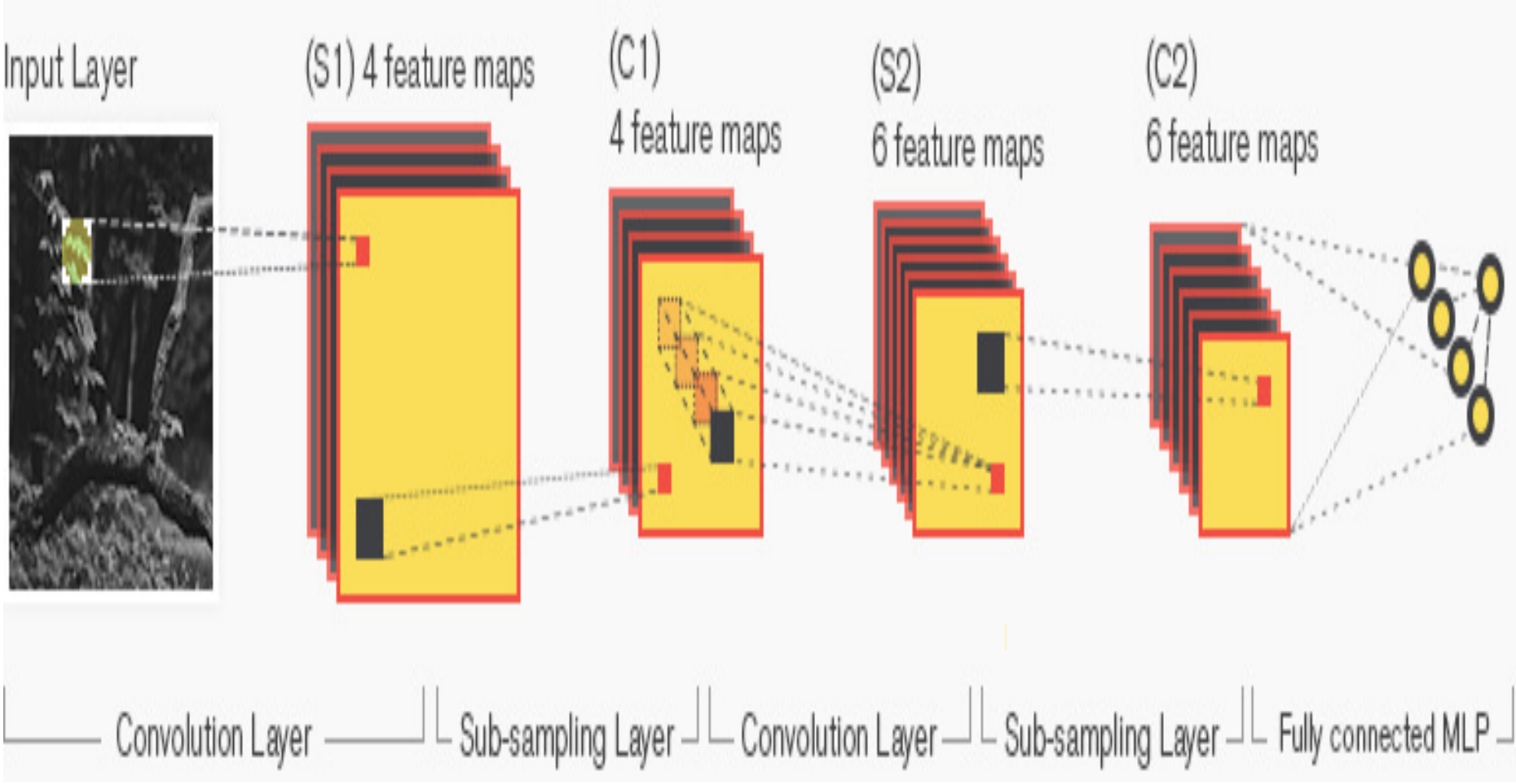


•In simple terms Deep Learning, is learning non linear functions with a net of many layers.

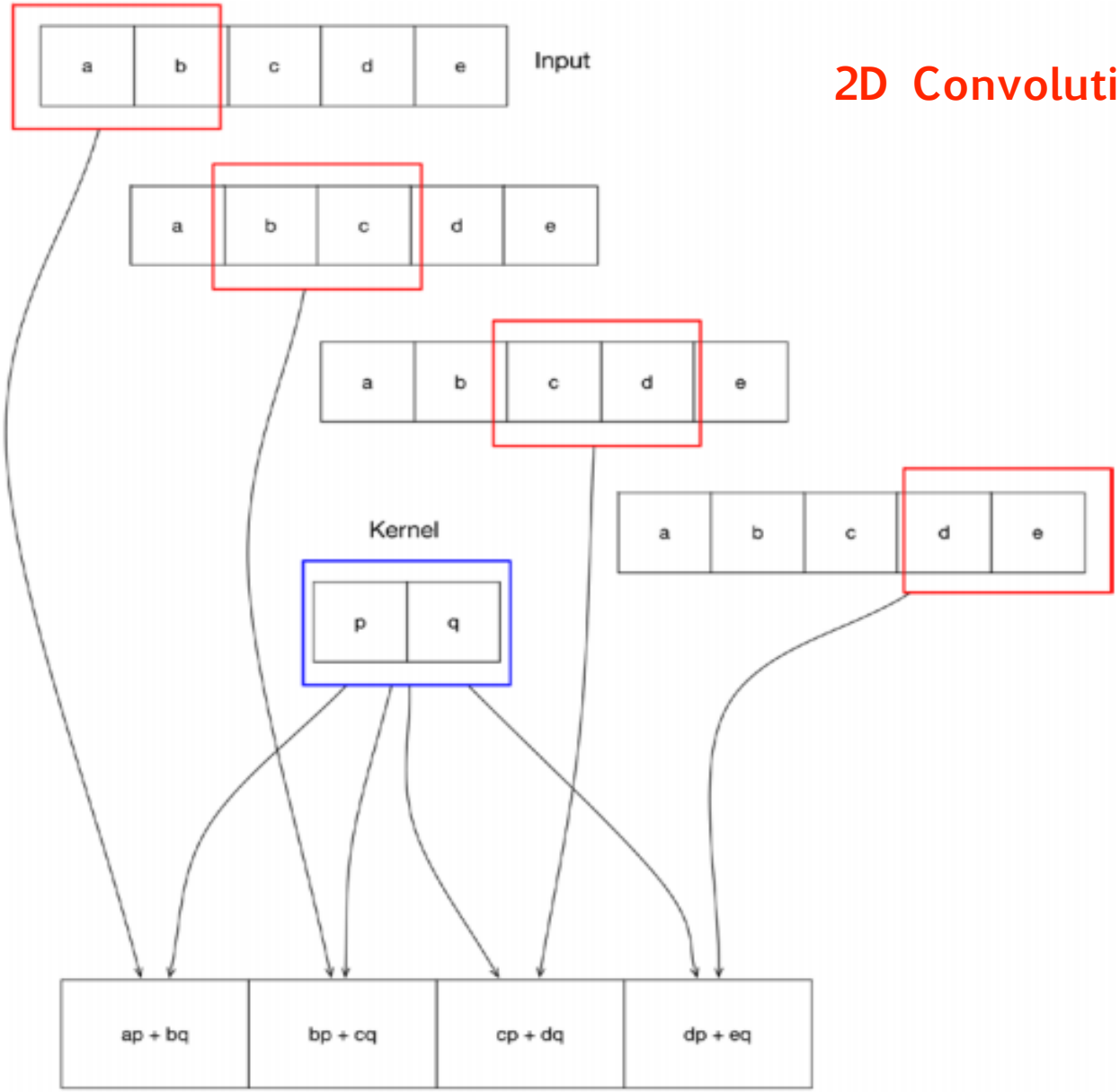
• Advanced algorithms and increased computing power like GPUs, have made use of this Technology possible.

• It may takes days to train Deep Nets on large data sets on low end hardware.

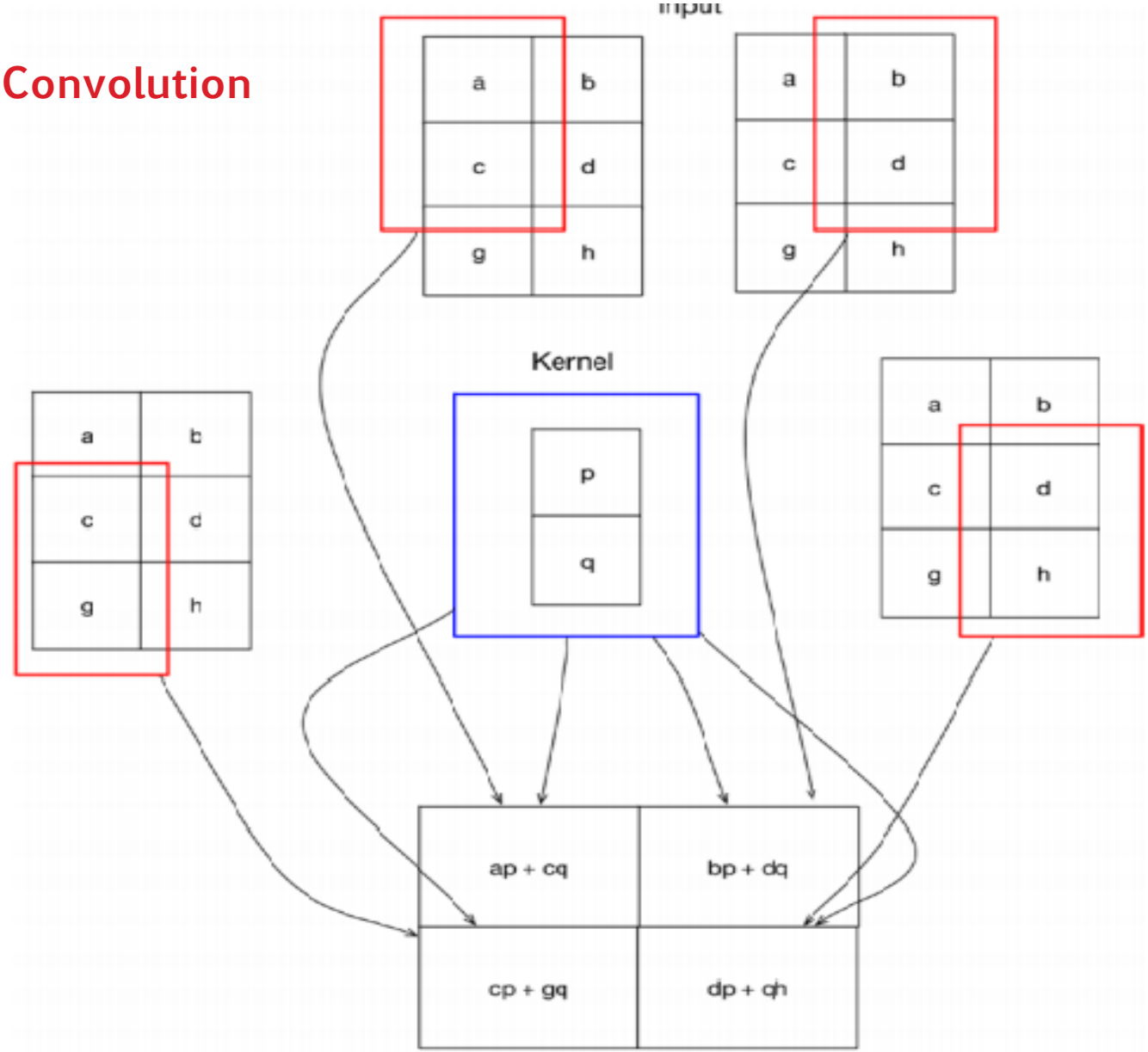
Deep Learning



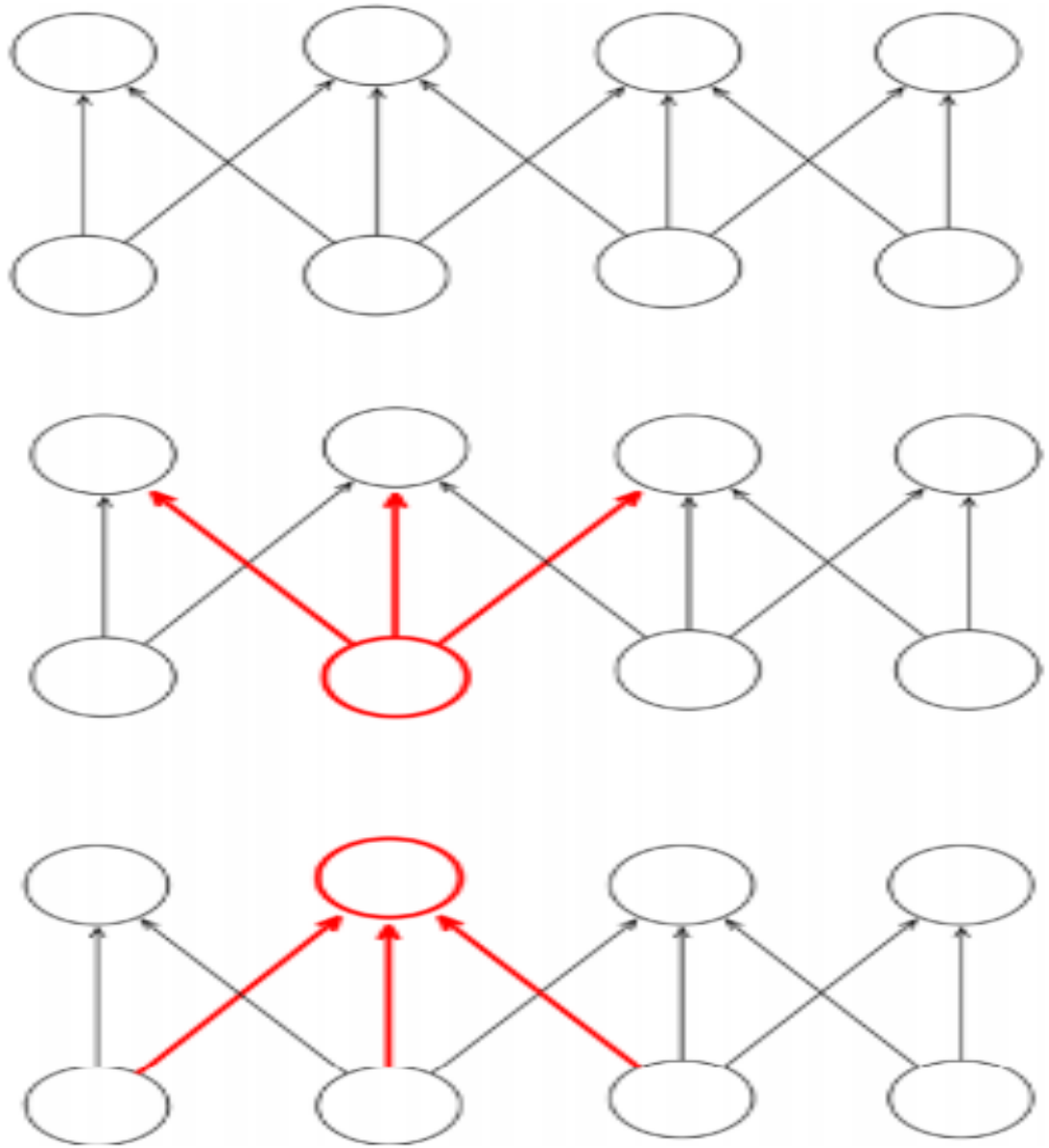
2D Convolution



2D Convolution

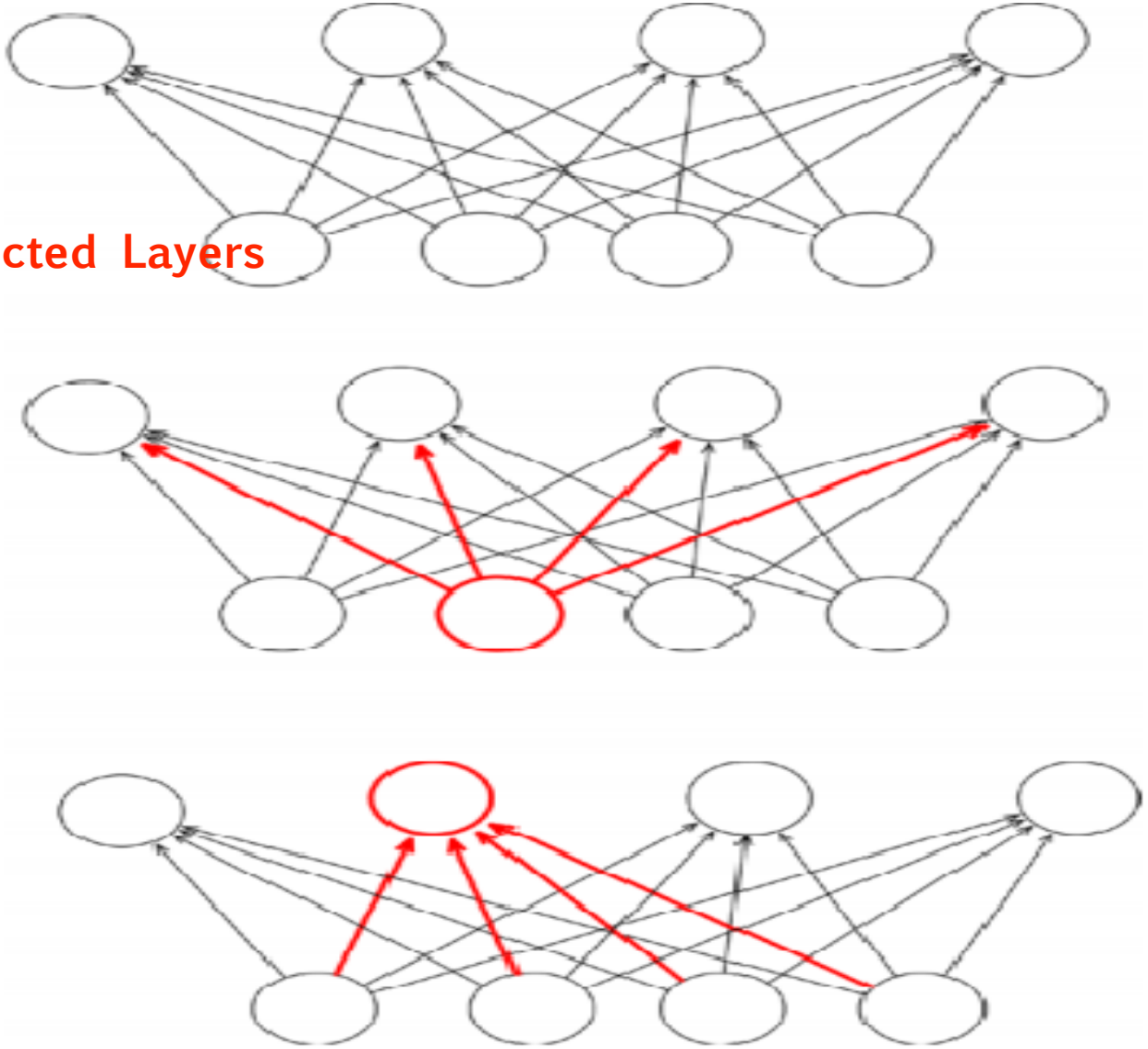


Convolution



Sparse Interactions in Convolution Layer

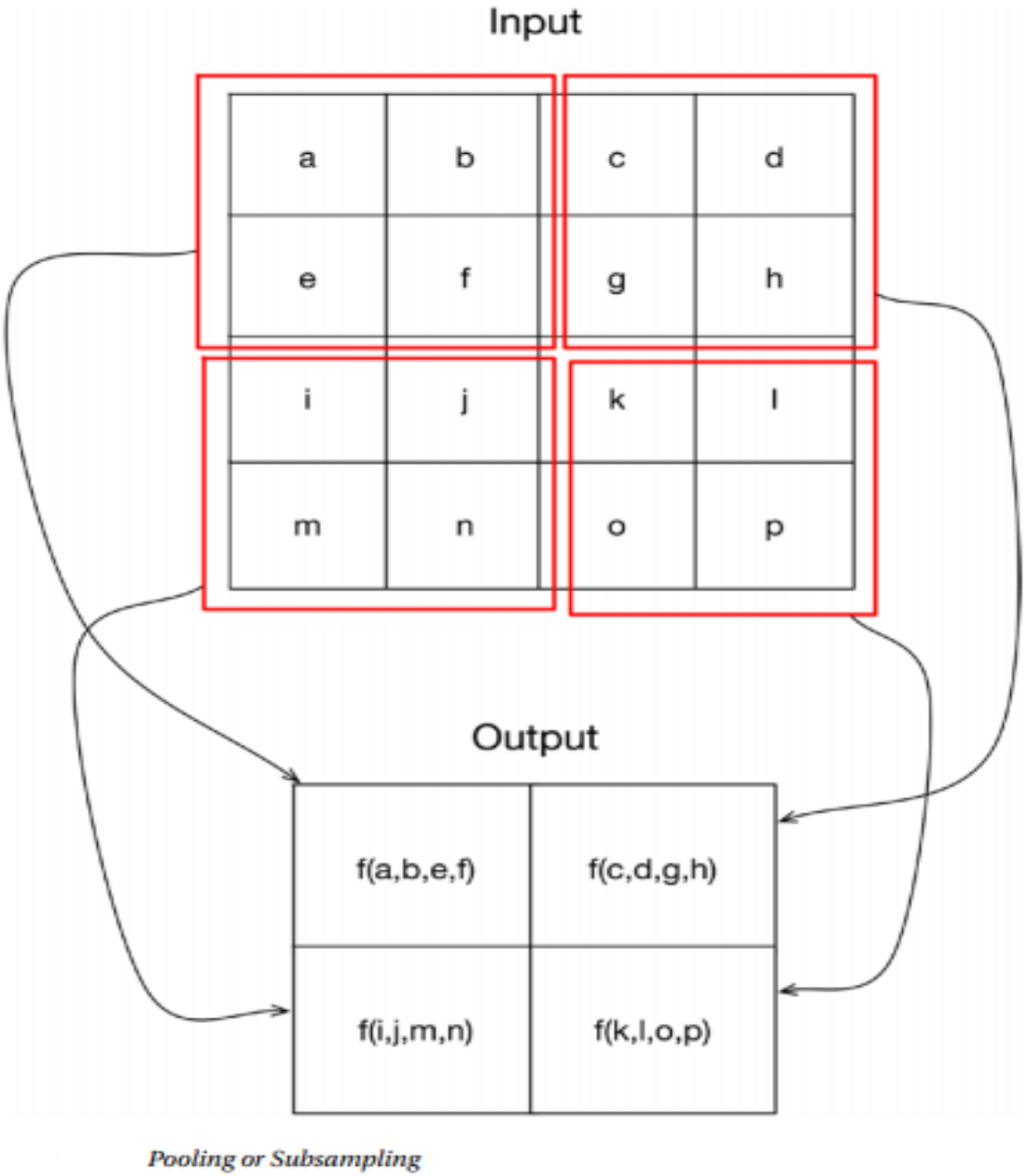
Fully Connected Layers



Dense Interactions in Fully Connected Layers

Pooling.

Ex: Maxpooling

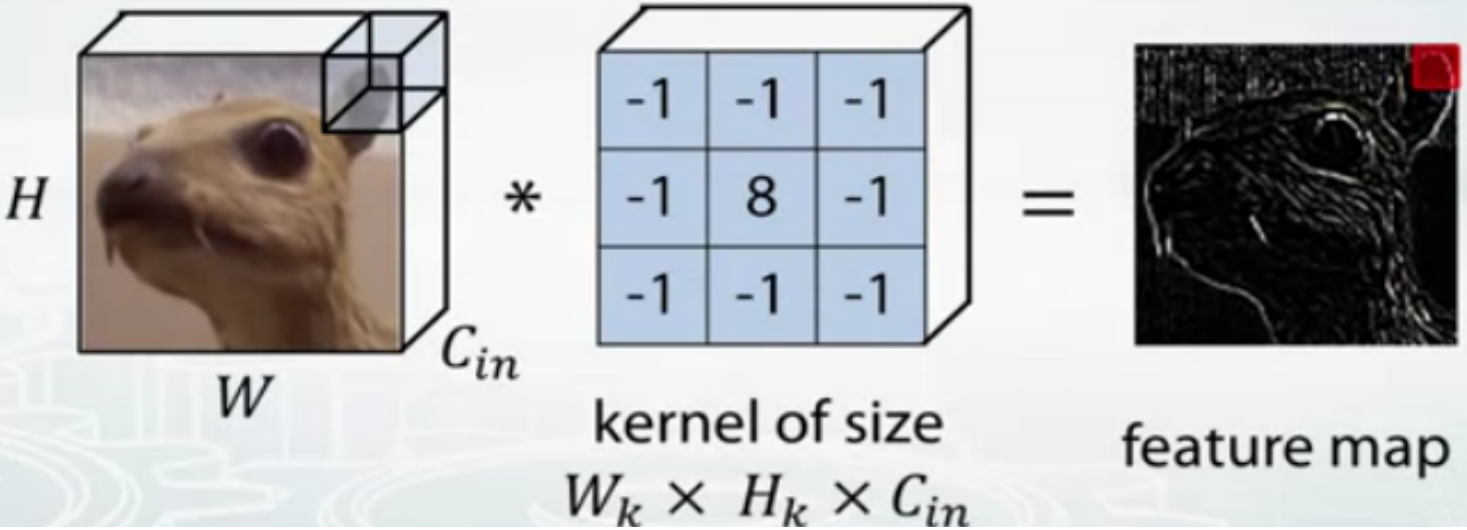


Pooling or Subsampling

Convolution

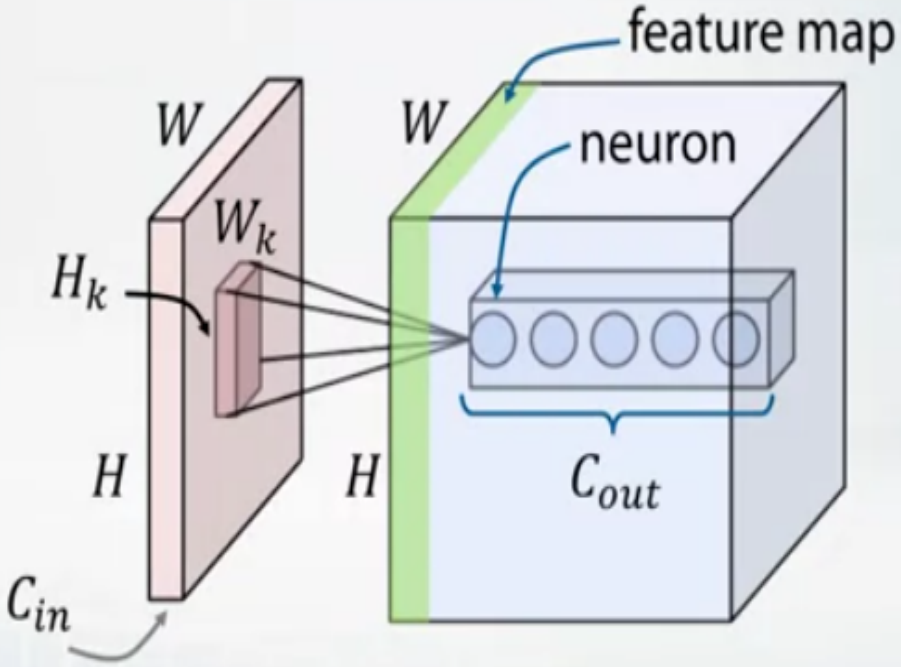
Let's say we have a color image as an input, which is $W \times H \times C_{in}$ **tensor** (multidimensional array), where

- W – is an image width,
- H – is an image height,
- C_{in} – is a number of input channels (e.g. 3 RGB channels).



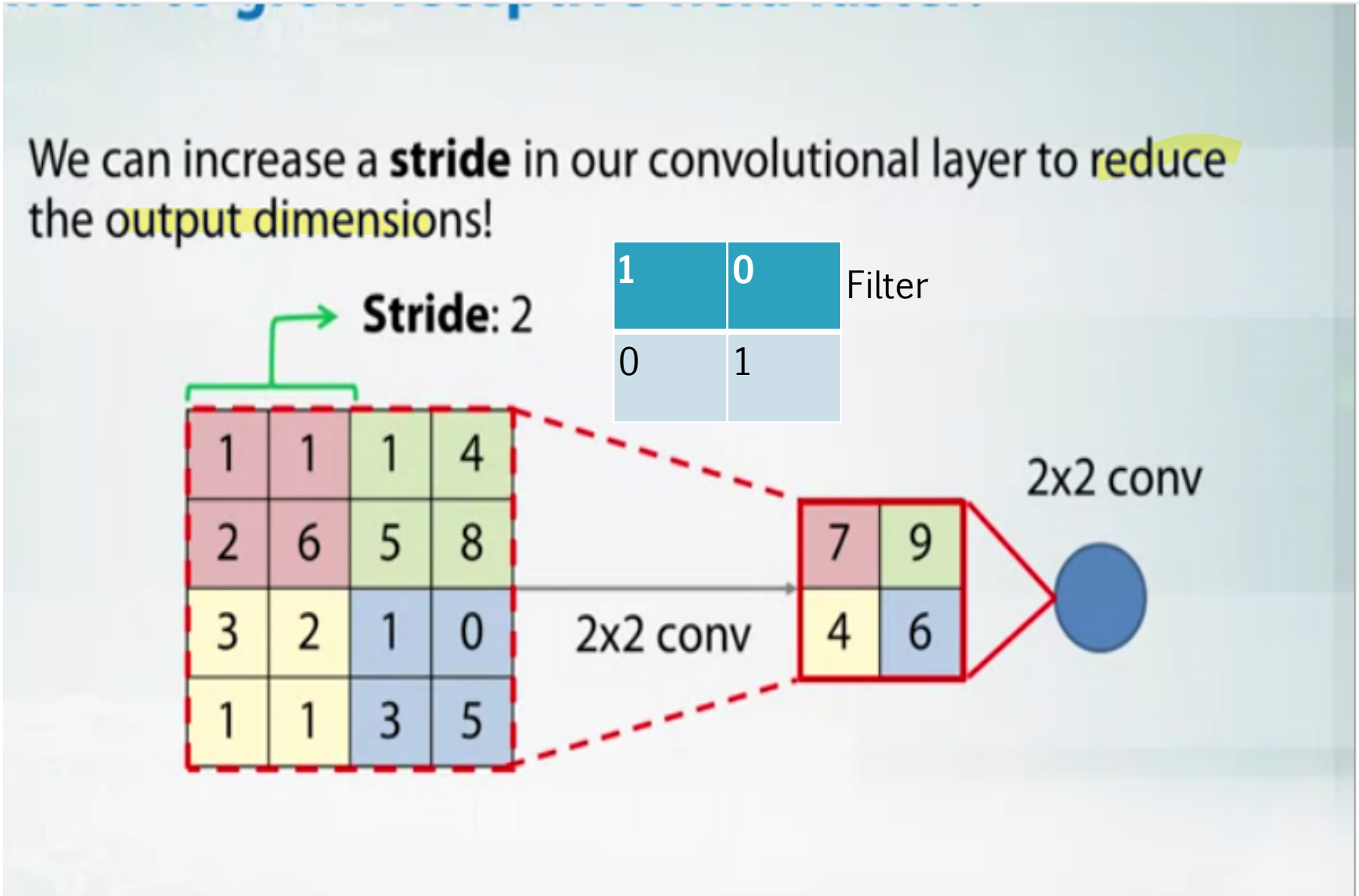
Filter Banks of C_{out}

- We want to train C_{out} kernels of size $W_k \times H_k \times C_{in}$.
- Having a stride of 1 and enough zero padding we can have $W \times H \times C_{out}$ output neurons.

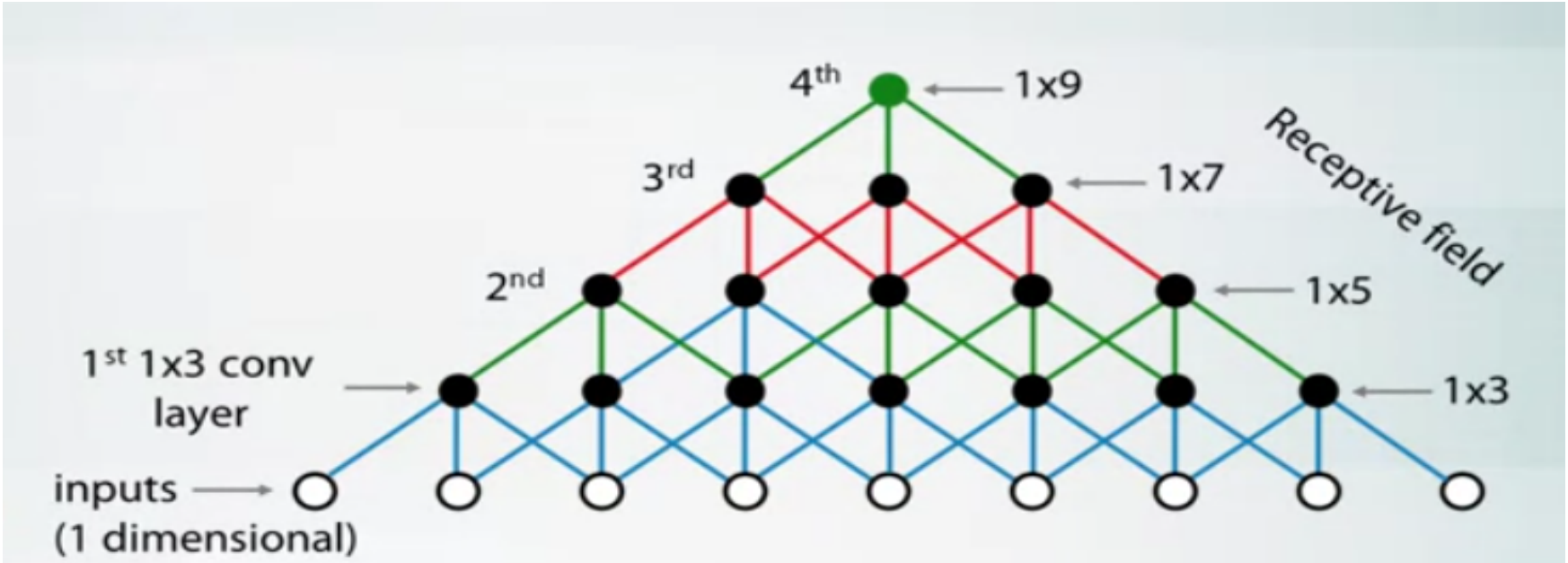


- Using $(W_k * H_k * C_{in} + 1) * C_{out}$ parameters.

Stride


We can increase a **stride** in our convolutional layer to reduce the output dimensions!
Stride: 2
1 0
0 1
Filter
1 1 1 4
2 6 5 8
3 2 1 0
1 1 3 5
2x2 conv
7 9
4 6
2x2 conv

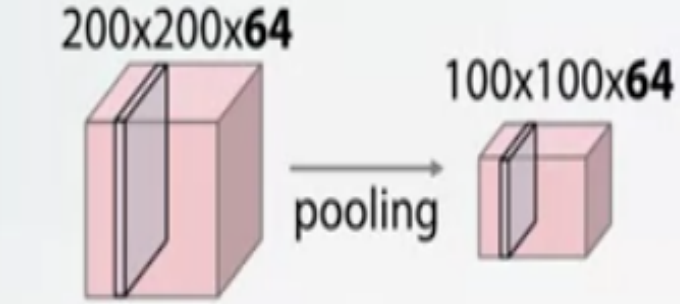
Increased Receptive Fiels/Spatial reach with repeated convolution

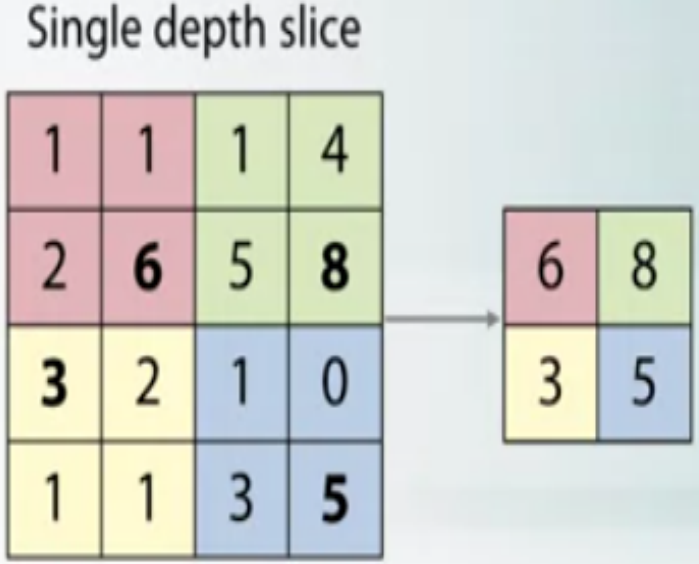


- If we stack N convolutional layers with the same kernel size 3x3 the receptive field on N-th layer will be $\boldsymbol{2N+1 \times 2N+1}$
- It looks like we need to stack a lot of convolutional layers! To be able to identify objects as big as the input image **300x300** we will need **150** convolutional layers!

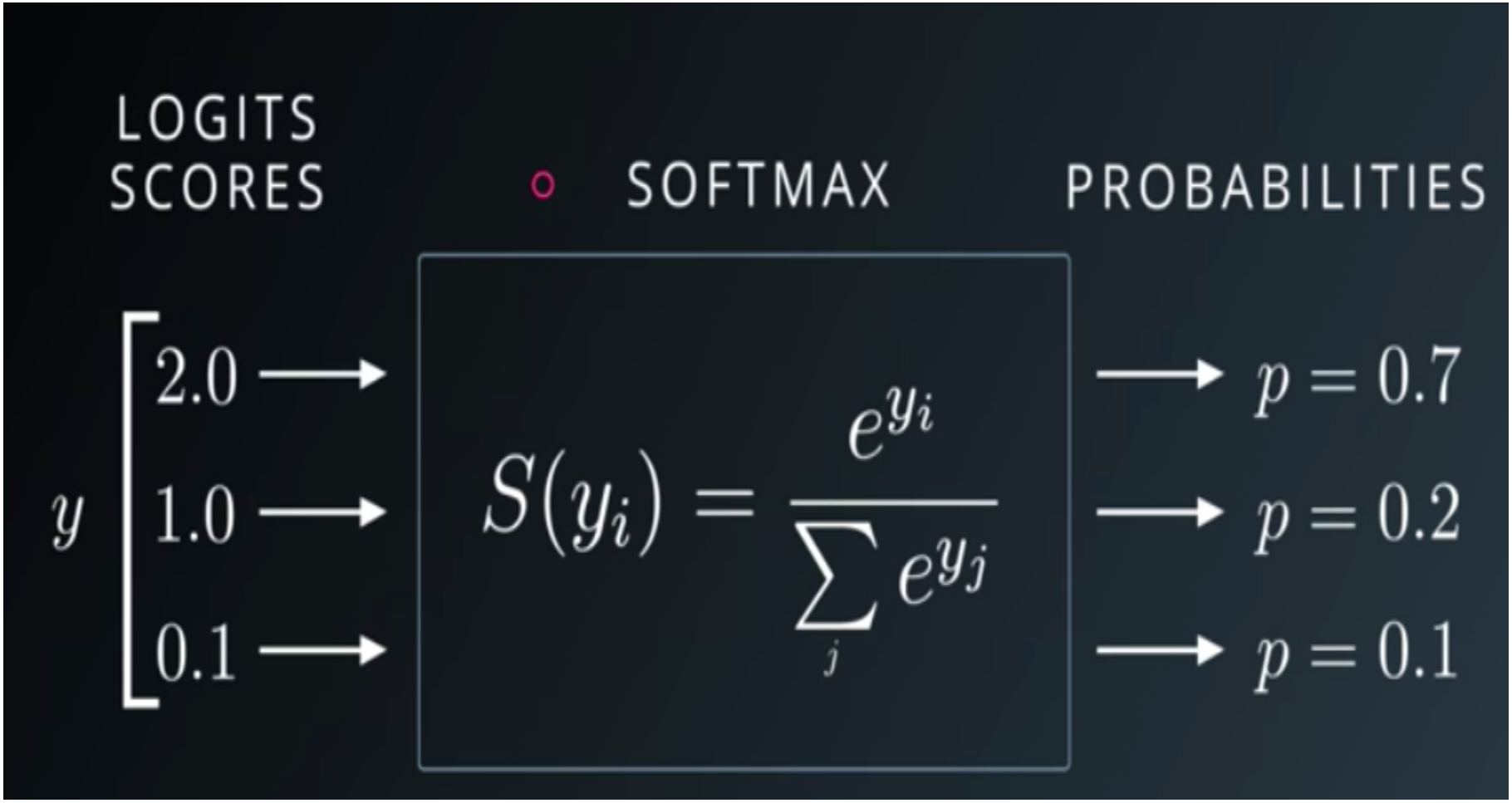
MaxPool Layer

This layer works like a convolutional layer but doesn't have kernel, instead it calculates **maximum** or **average** of input patch values.

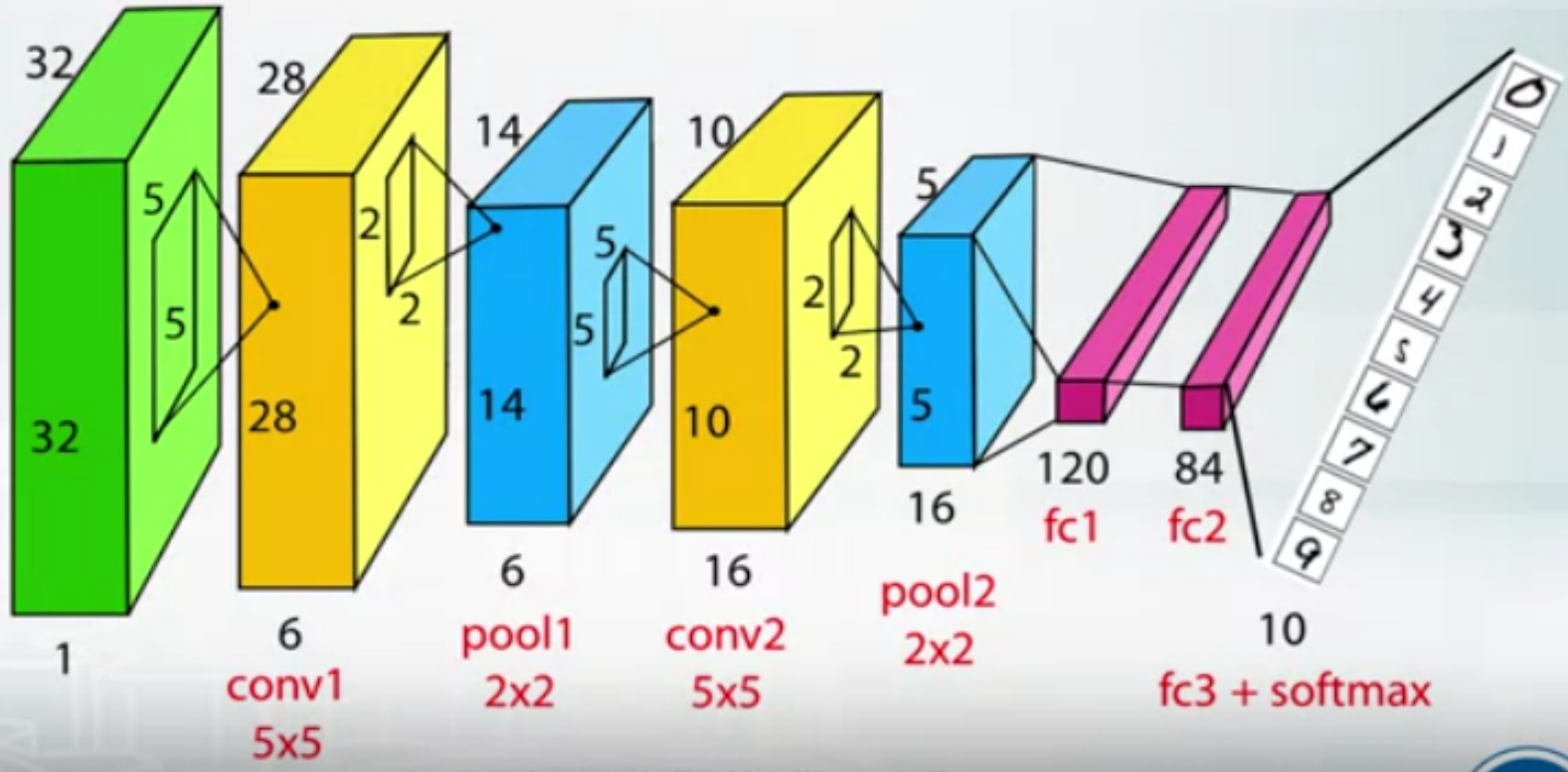




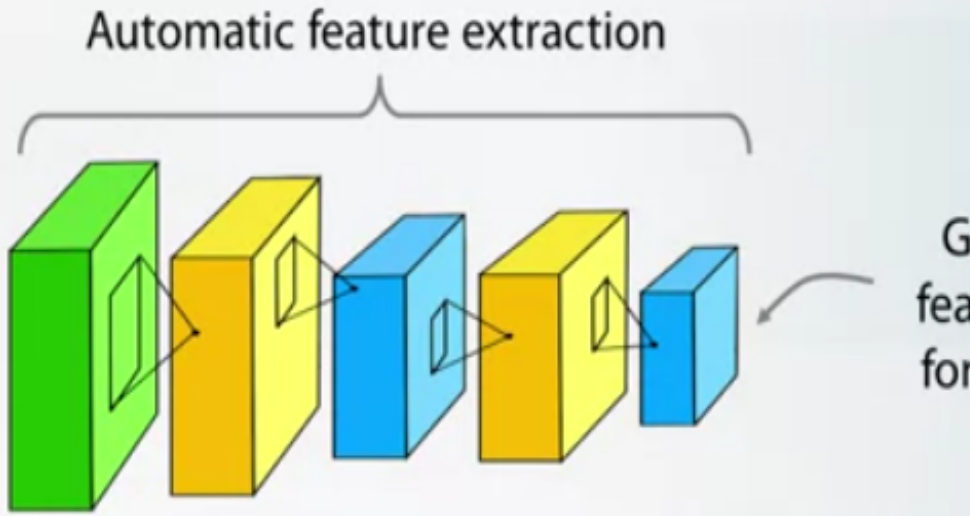
Softmax


LOGITS
SCORES
SOFTMAX
PROBABILITIES
y
2.0
1.0
0.1
$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$
$p = 0.7$
$p = 0.2$
$p = 0.1$

LeNet-5 architecture (1998) for handwritten digits recognition on MNIST dataset:



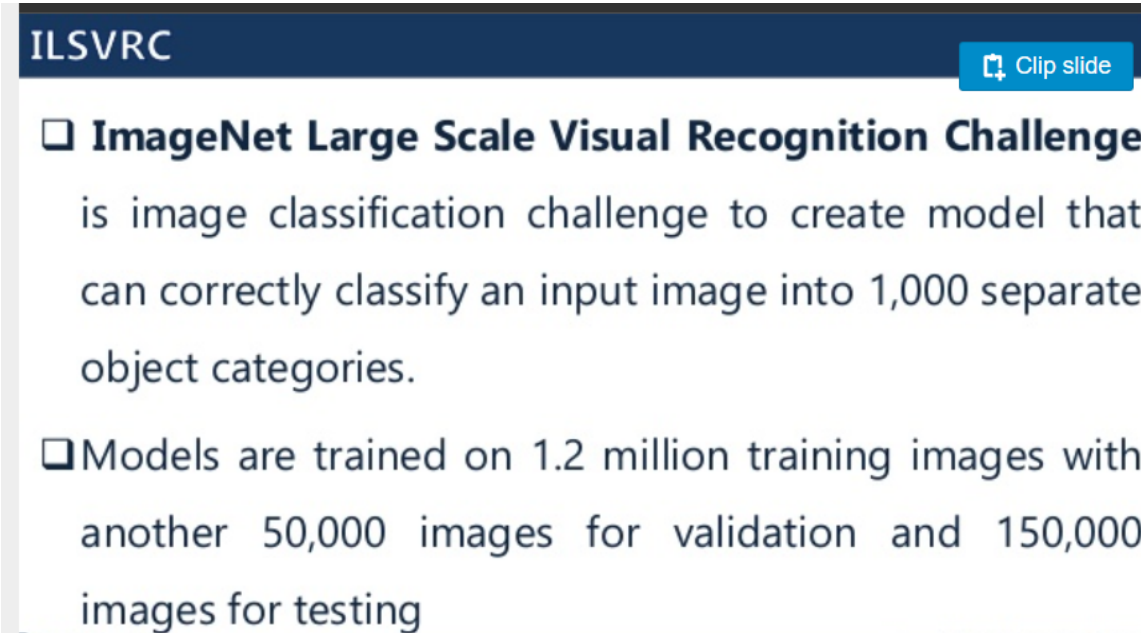
Neurons of deep convolutional layers learn complex representations that can be used as features for classification with MLP.



Inputs that provide highest activations:

ImageNet Challenge
CNN /Deep Learning solutions

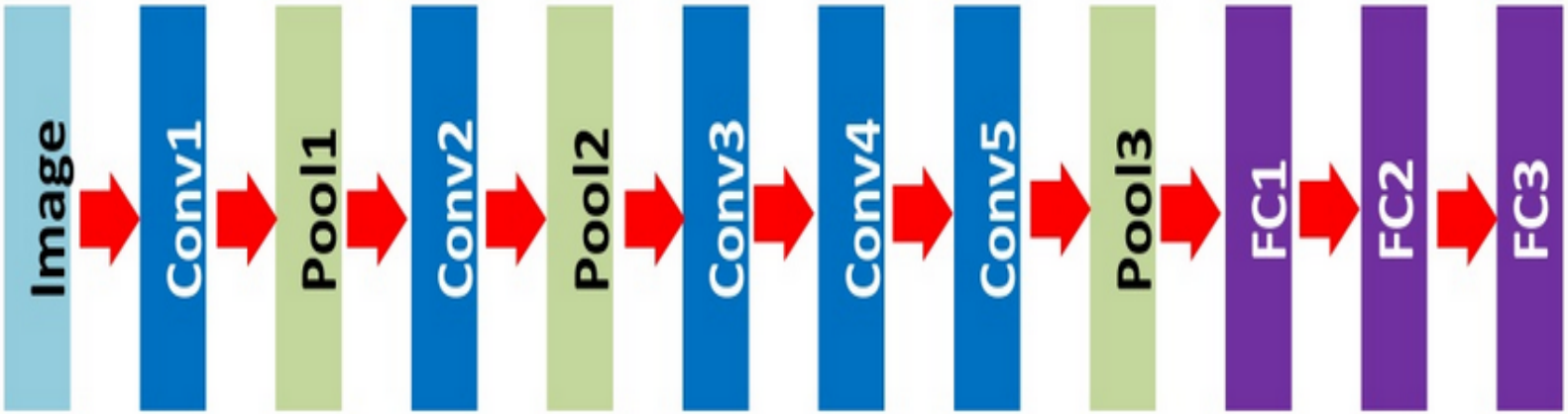
- ILSVRC
- AlexNET (2012)
- VGGNET (2014)
- googleNet (2014)
- RESNET (2015)



Objects in ImageNet Dataset

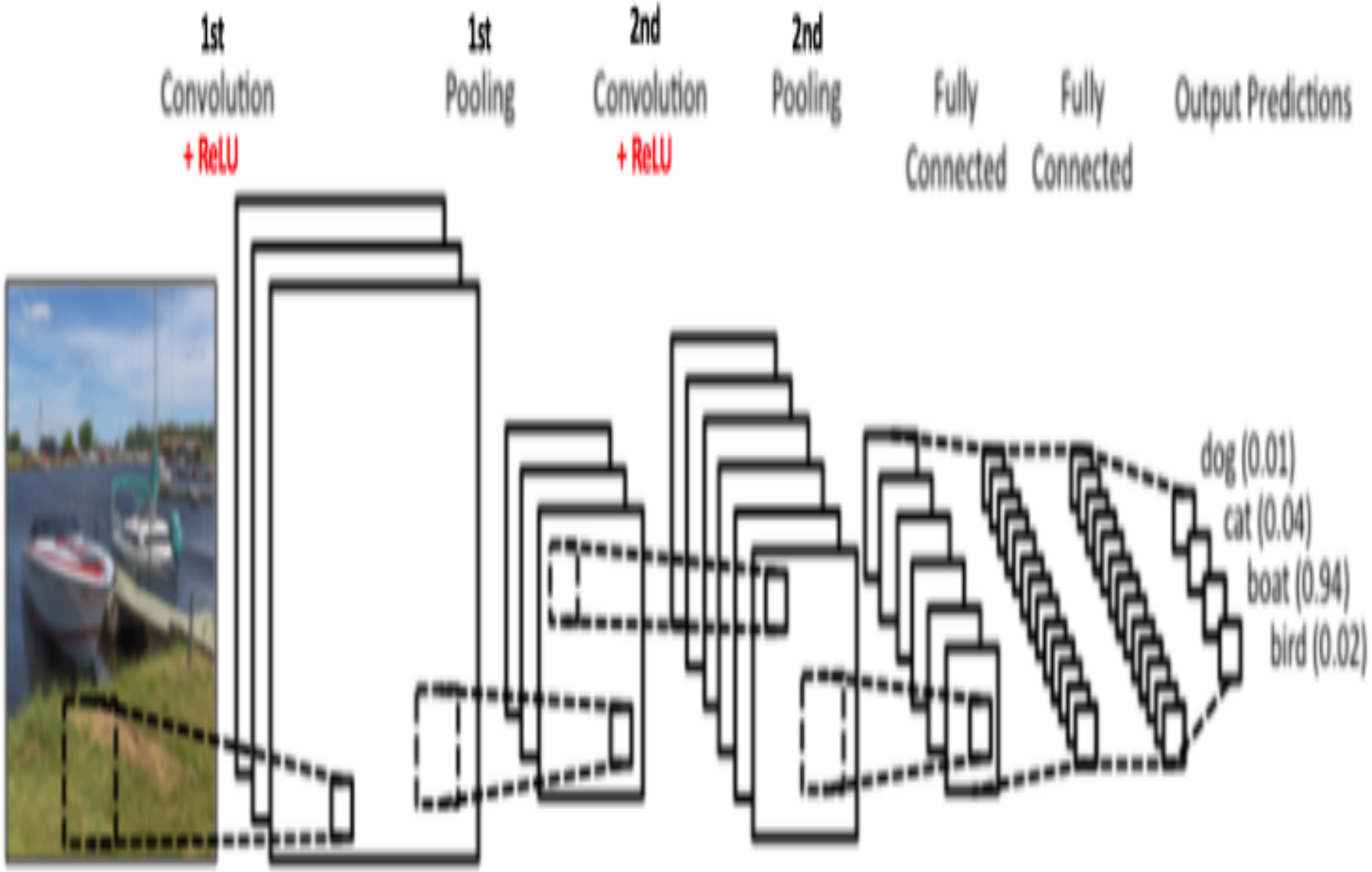
AlexNet (2012)

- **AlexNet** has 8 layers without count pooling layers.
- **AlexNet** use ReLU for the nonlinearity functions
- **AlexNet** trained on two GTX 580 GPUs for **five to six days**



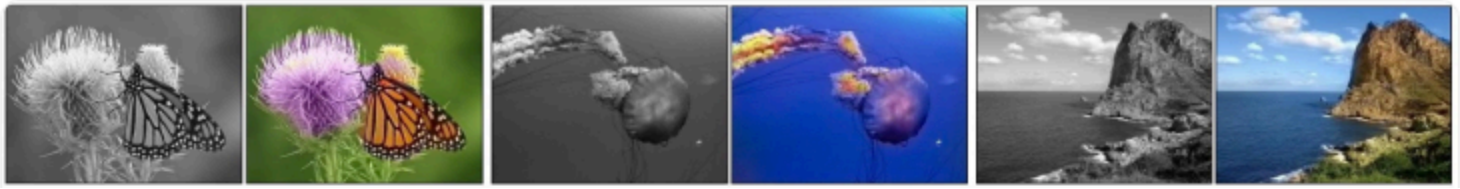
ResNet (2015)

- **ResNet** the winner of the competition ILSVRC 2015 with **3.6%** Top-5 error rate.
- **ResNet** mainly inspired by the philosophy of VGGNet.
- **ResNet** proposed a residual learning approach to ease the difficulty of training deeper networks. Based on the design ideas of Batch Normalization (BN), small convolutional kernels.
- **ResNet** is a new 152 layer network architecture.
- **ResNet** Trained on an 8 GPU machine for **two to three weeks**


1st
Convolution
+ ReLU
1st
Pooling
2nd
Convolution
+ ReLU
2nd
Pooling
Fully
Connected
Fully
Connected
Output Predictions
dog (0.01)
cat (0.04)
boat (0.94)
bird (0.02)

Deep Learning Applications

- Automatic Image Colouring



- **Automatic Machine Translation**

- **Object Classification and Detection in Photographs**

ImageNet from Alex Krizhevsky

Deep Learning Applications (Contd)

- **Automatic music, script, … generation**

- **Automatic Game Playing**
  - From pixels on screen to Game moves (Google DeepMind)
- **Self-driving cars**
- **Deep Learning in Healthcare (predictive care)**
- **Voice Search & Voice-Activated Assistants**
- **….**

Convolutional Neural Networks

- Convolution Neural Networks (CNNs) in essence are neural networks that employ the convolution operation (instead of a fully connected layer) as one of its layers.
- Applied to problems wherein the input data on which predictions are to be made has a known **grid like topology** **like a time series** (which is a 1-D grid) or **an image** (which is a 2-D grid).

Convolution Operation

- convolution operation in one dimension. Given an input I(t) and a kernel K(a) the convolution operation is given by

$$s(t) = \sum_a I(a) \cdot K(t-a)$$

An equivalent form of this operation given commutativity of the convolution operation is as follows:

$$s(t) = \sum_a I(t-a) \cdot K(a)$$

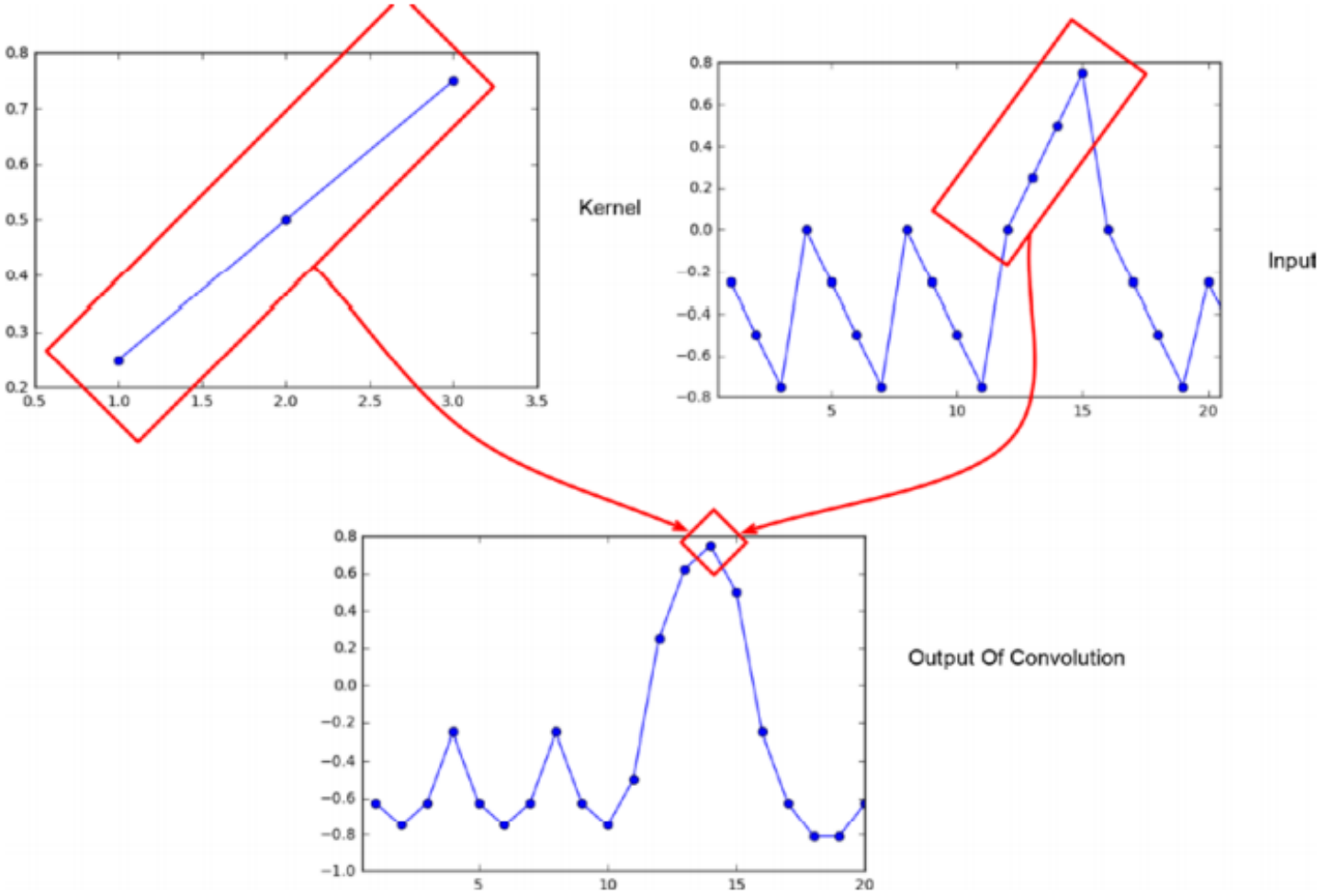
Furthermore, the negative sign (flipping) can be replaced to get cross-correlation given as follows:

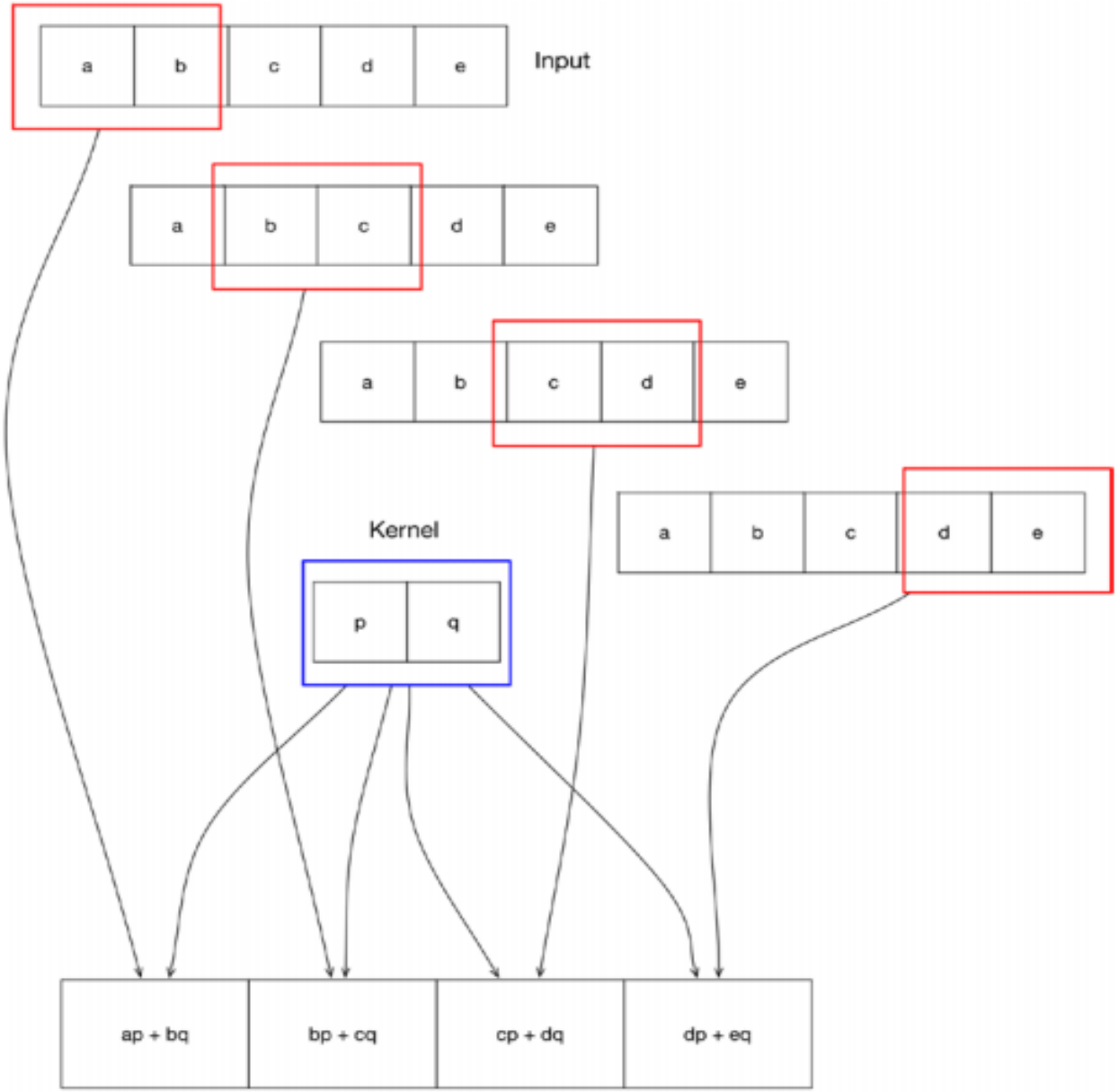
$$s(t) = \sum_a I(t+a) \cdot K(a)$$

Convolution Operation

- The essence of the operation is that the Kernel is a much shorter set of data points as compared to the input, and the output of the convolution operation is higher when the input is similar to the kernel. Figures in following slides illustrate this key idea. We take an arbitrary input and an arbitrary kernel, perform the convolution operation, and the highest value is achieved when the kernel is similar to a particular portion of the input.

Convolution Operation




a
b
c
d
e
Input
a
b
c
d
e
a
b
c
d
e
Kernel
p
q
a
b
c
d
e
ap + bq
bp + cq
cp + dq
dp + eq

Following are the points to be noted for convolution operation

1. The input is an arbitrary large set of data points.
2. The Kernel is a set of data points smaller in number to the input.
3. The convolution operation in a sense slides the kernel over the input and computes how similar the kernel is with the portion of the input.
4. The convolution operation produces the highest value where the Kernel is most similar with a portion of the input.

Convolution Operation

The convolution operation can be extended to two dimensions. Given an input $I(m,n)$ and a kernel $K(a,b)$ the convolution operation is given by

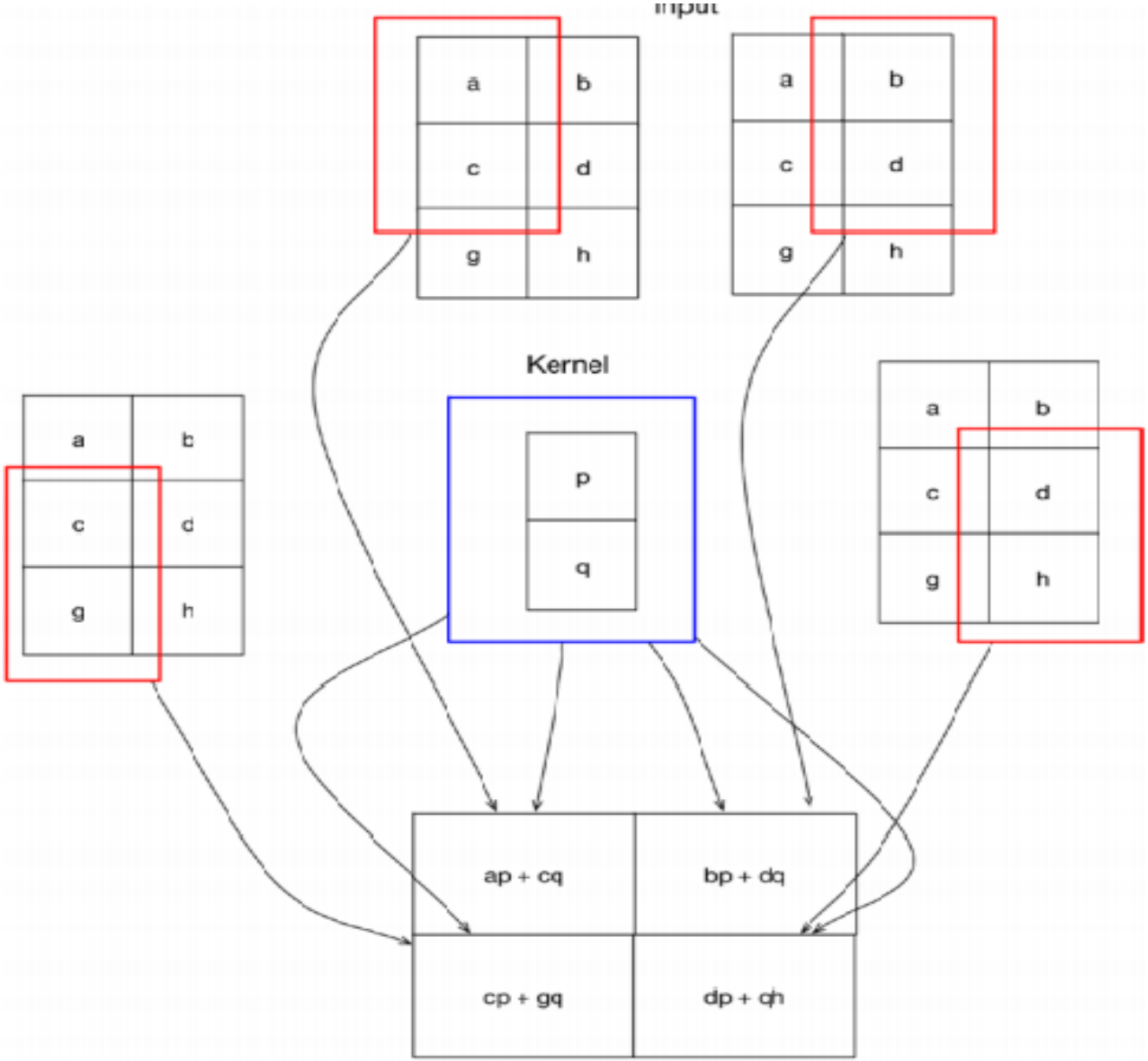
$$s(t)=\sum_a\sum_b I(a,b)\cdot K(m-a,n-b)$$

An equivalent form of this operation given commutativity of the convolution operation is as follows:

$$s(t)=\sum_a\sum_b I(m-a,n-b)\cdot K(a,b)$$

Furthermore, the negative sign (flipping) can be replaced to get cross-correlation given as follows:

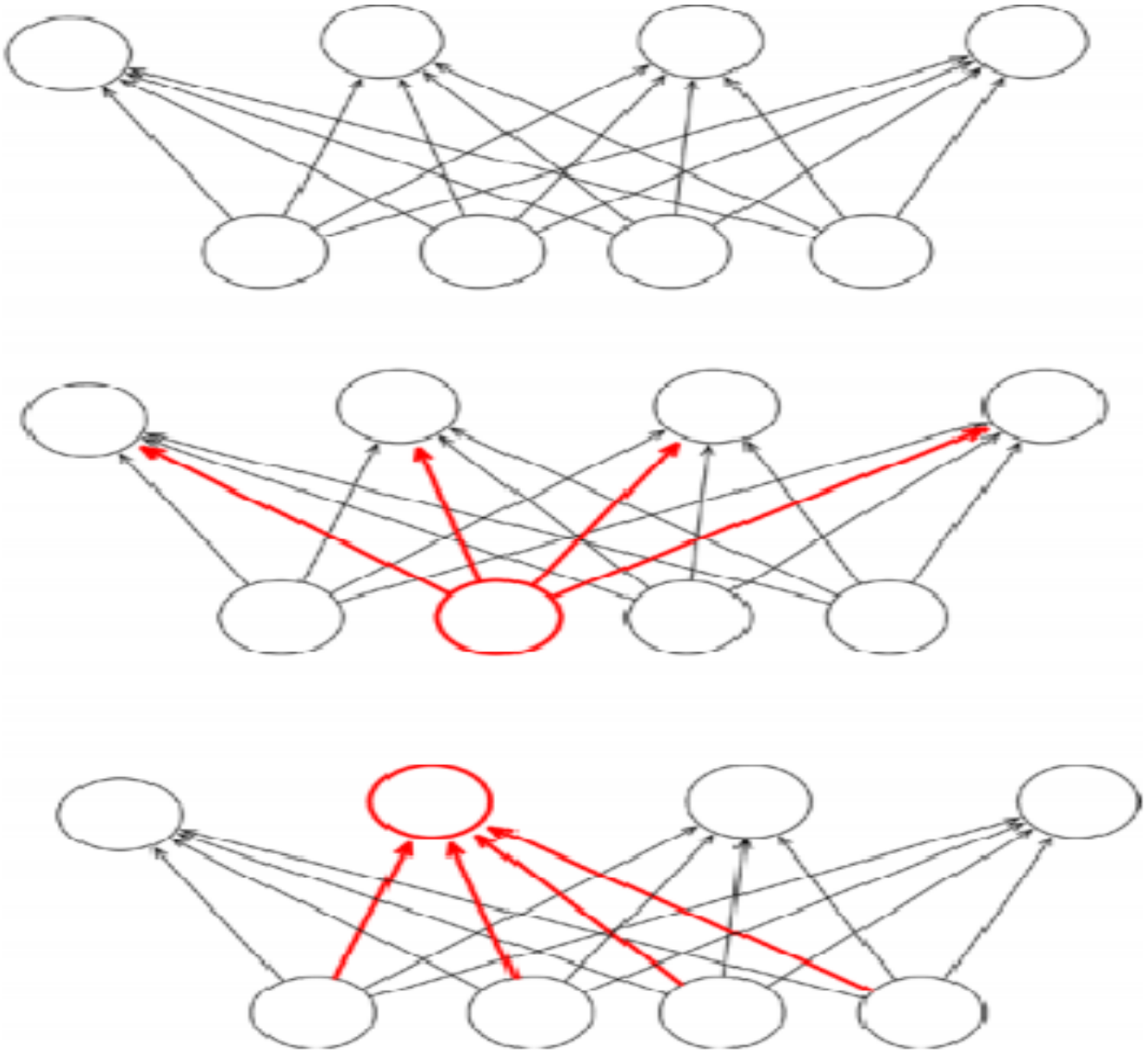
$$s(t)=\sum_a\sum_b I(m+a,n+b)\cdot K(a,b)$$


Input
a
b
c
d
g
h
Kernel
p
q
ap + cq
bp + dq
cp + gq
dp + qh

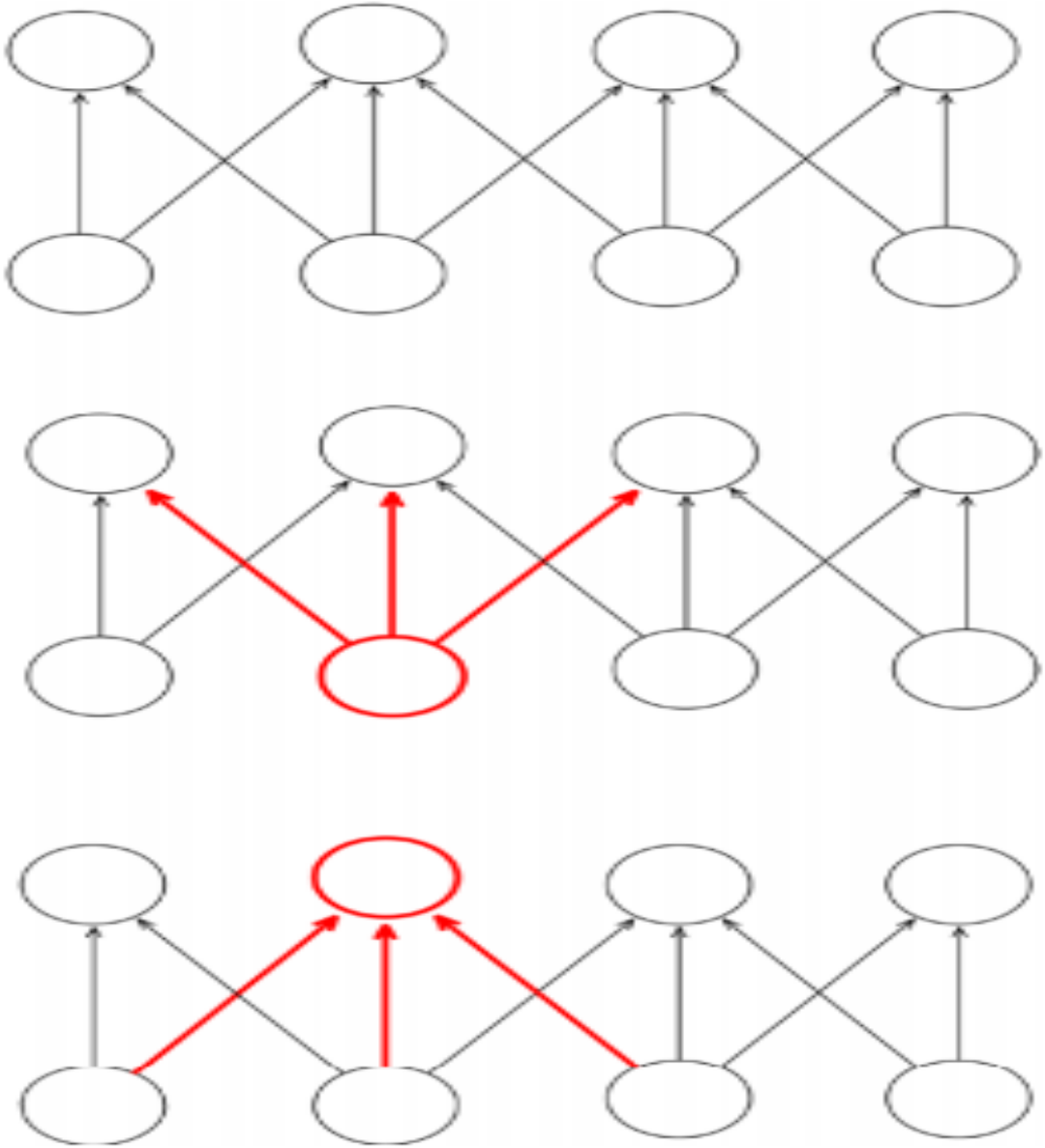
- Having introduced the convolution operation, we can now dive deeper into the key constituent parts of a CNN, were a convolution layer is used instead of a fully connected layer which involves a matrix multiplication. So, a fully connected layer can be described as y = f(x.w) where x is the input vector, y is the output vector, w is a set of weights, and f is the activation function. Correspondingly, a convolution layer can be described as y = f(s(x.w))where **s denotes the convolution operation between the input and the weights.**

Difference b/w Convolution and fully connected layers

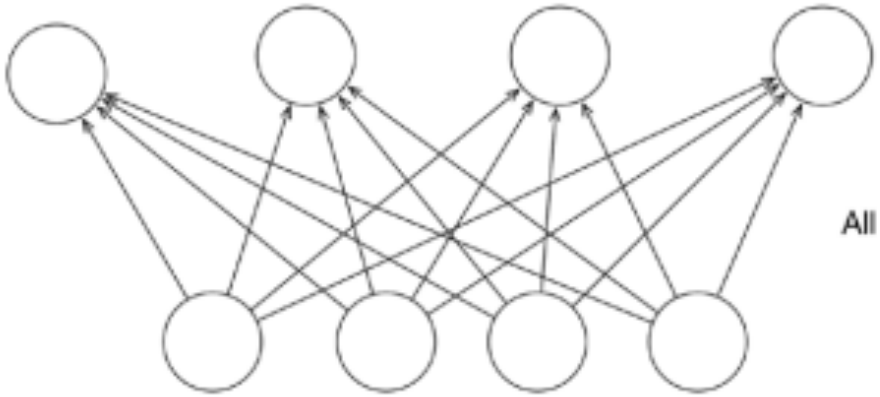
1. For the same number of inputs and outputs, the fully connected layer has a lot more connections, and correspondingly weights that a convolution layer.
2. The interactions amongst inputs to produce outputs are fewer in convolution layers as compared to many interactions in the case of a fully connected layer. This is referred to as sparse interactions.
3. Parameters/weights are shared across the convolution layer, given that the kernel is much smaller than the input and the kernel slides across the input. Thus, there are a lot fewer unique parameters/weights in a convolution layer.



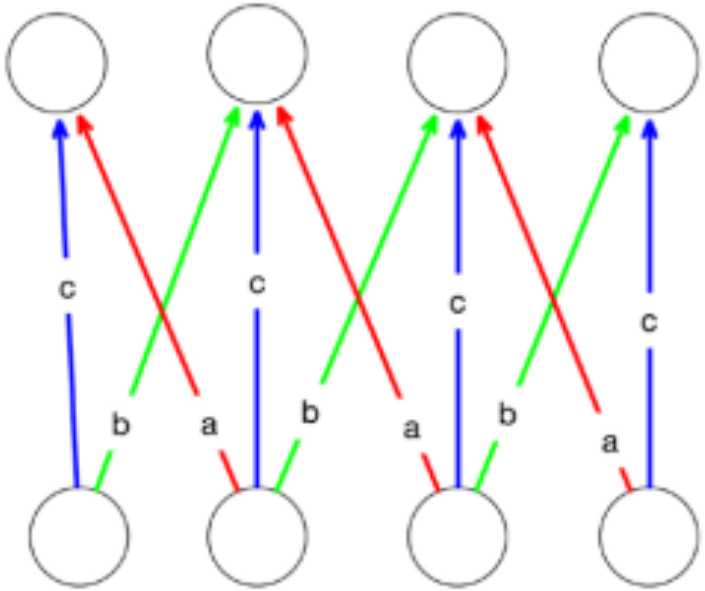
Dense Interactions in Fully Connected Layers



Sparse Interactions in Convolution Layer



All Unique Weights



Only 3 unique Weights

Parameter Sharing Tied Weights

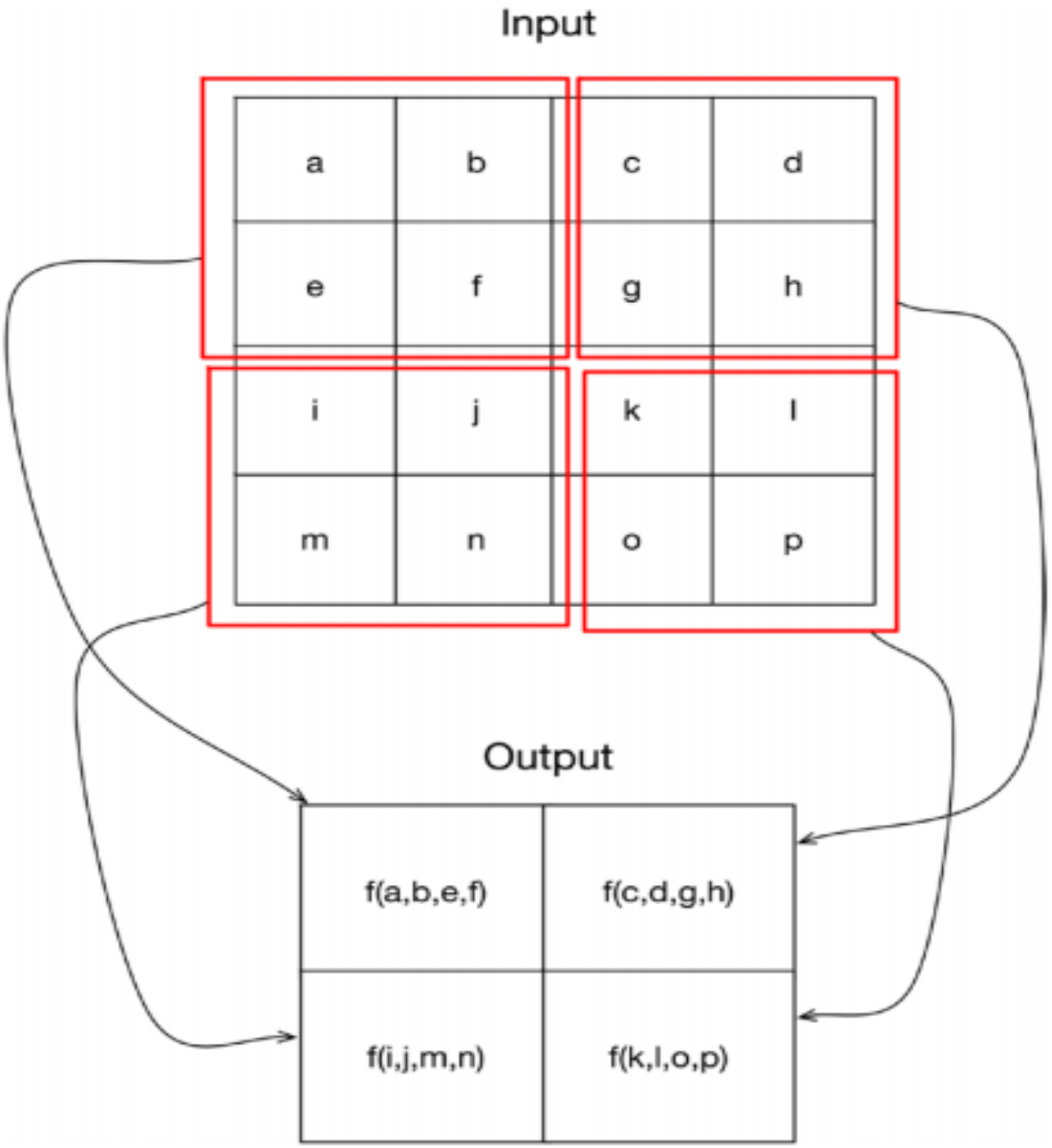
Pooling Operation

- For instance, assume that the task at hand is to learn to detect faces in photographs. Let us also assume that the faces in the photograph are tilted (as they generally are) and suppose that we have a convolution layer that detects the eyes. **We would like to abstract the location of the eyes in the photograph from their orientation. The pooling operation achieves this and is an important constituent of CNNs**.

Pooling Operation

- Figure illustrates the pooling operation for a two-dimensional input. The following points are to be noted:

1. Pooling operates over a portion of the input and applies a function f over this input to produce the output.
2. The function f is commonly the max operation (leading to max pooling), but other variants such as average or L2 norm can be used as an alternative.
3. For a two-dimensional input, this is a rectangular portion.
4. The output produced as a result of pooling is much smaller in dimensionality as compared to the input.

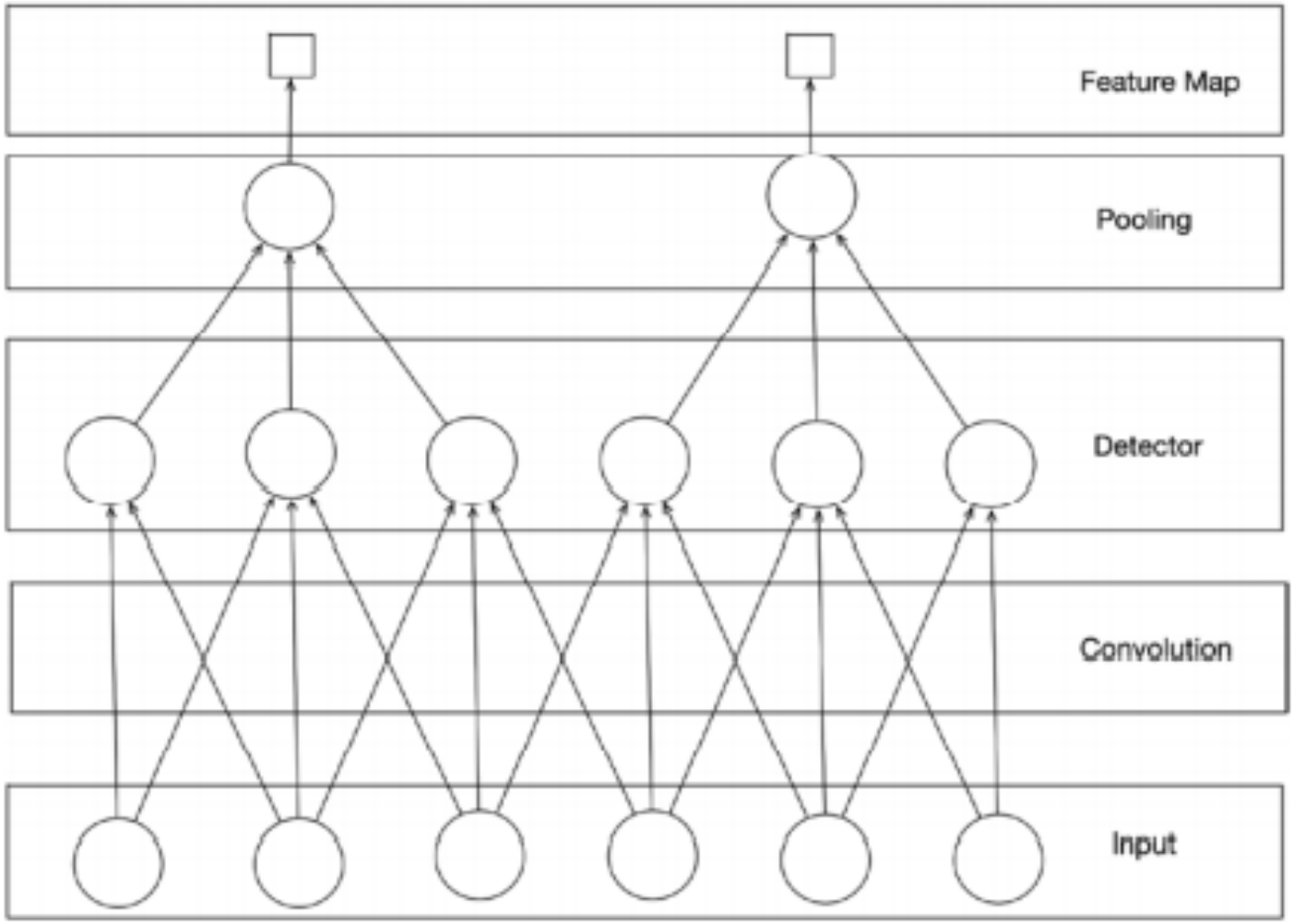


Pooling or Subsampling

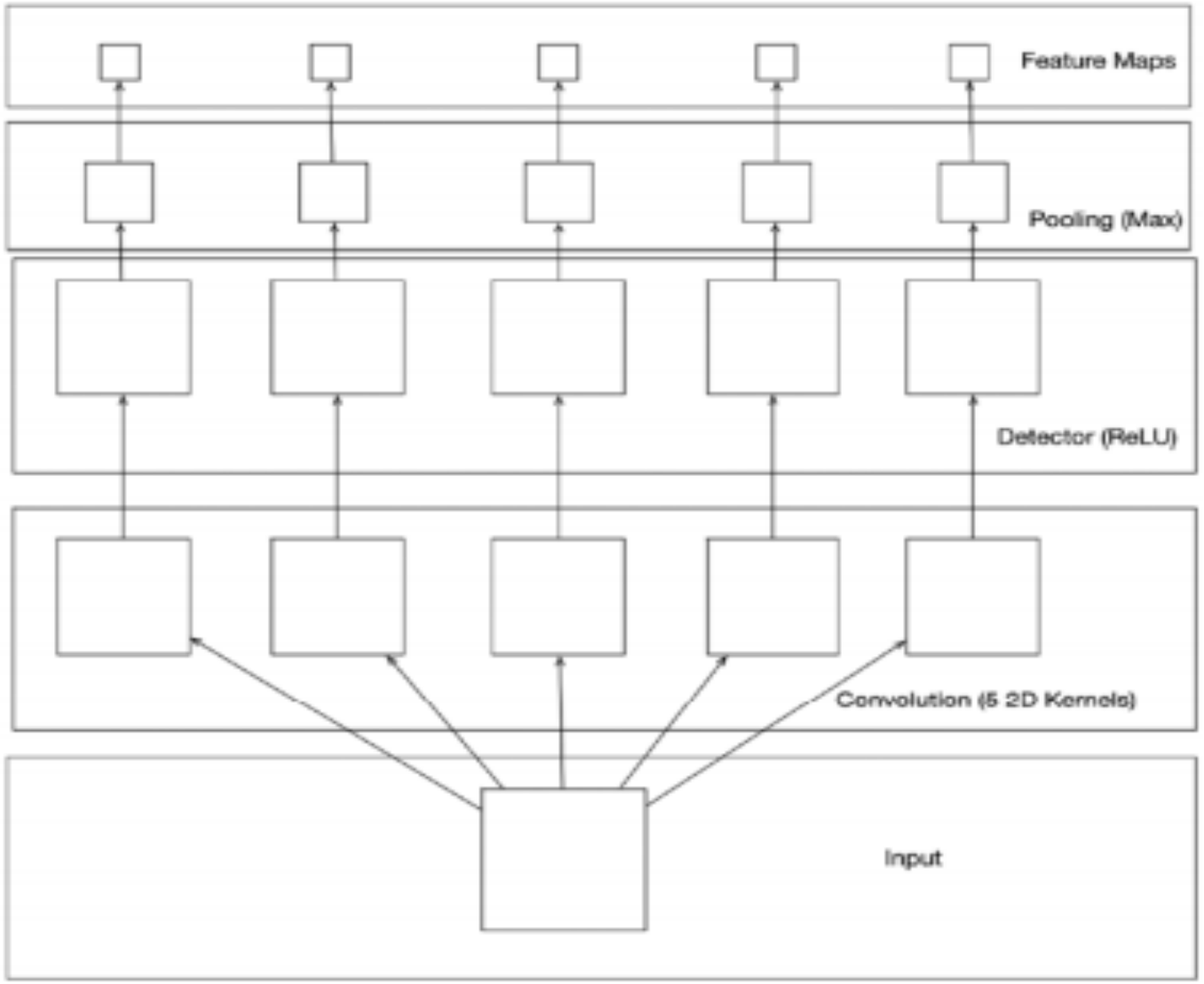
Convolution-Detector-Pooling Building Block

- The following points are to be noted:

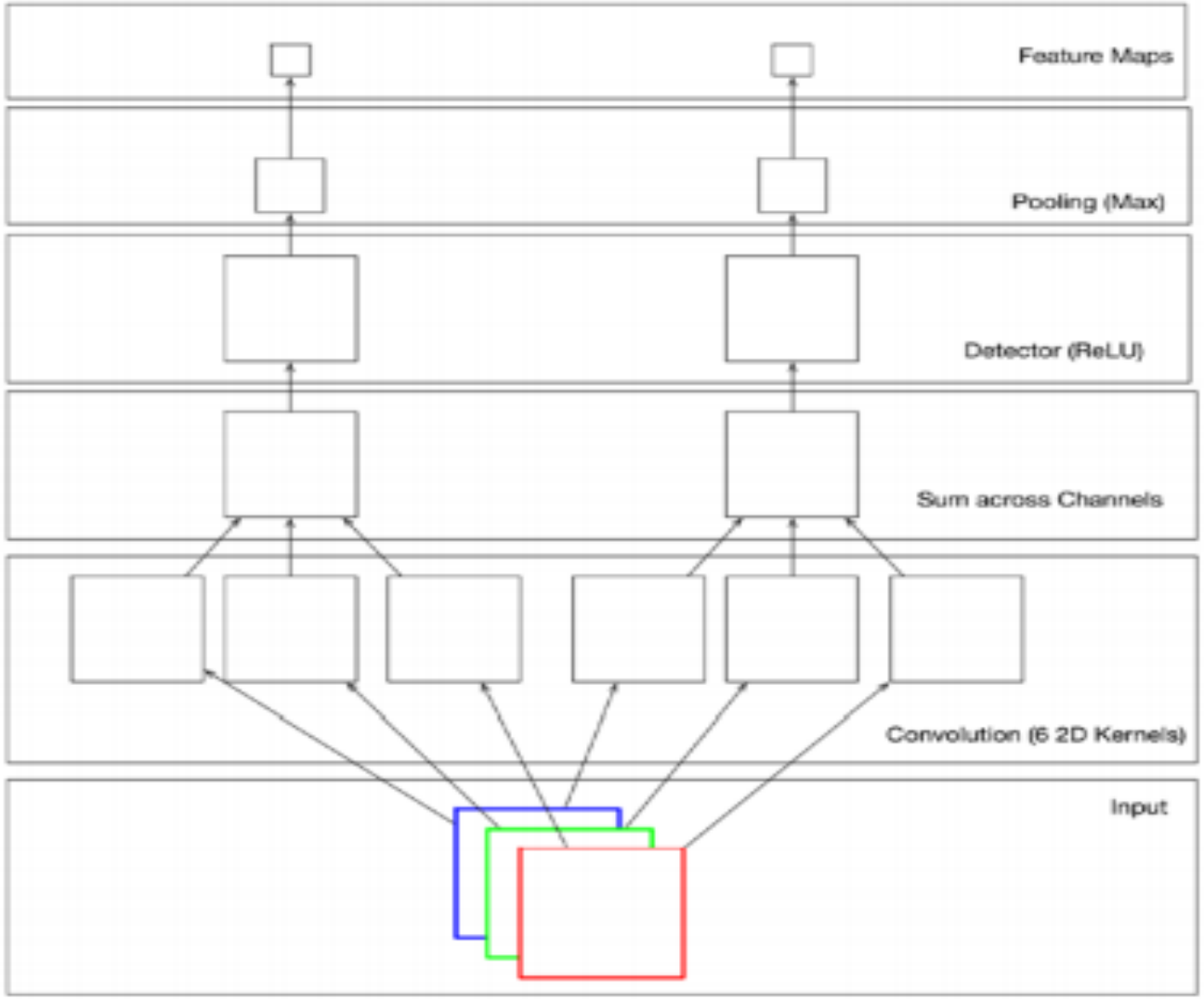
1. The detector stage is simply a non-linear activation function.
2. The convolution, detector, and pooling operations are applied in sequence to transform the input to the output. The output is referred to as a feature map.
3. The output typically is passed on to other layers (convolution or fully connected).
4. Multiple Convolution-Detector-Pooling blocks can be applied in parallel, consuming the same input and producing multiple outputs or feature maps.



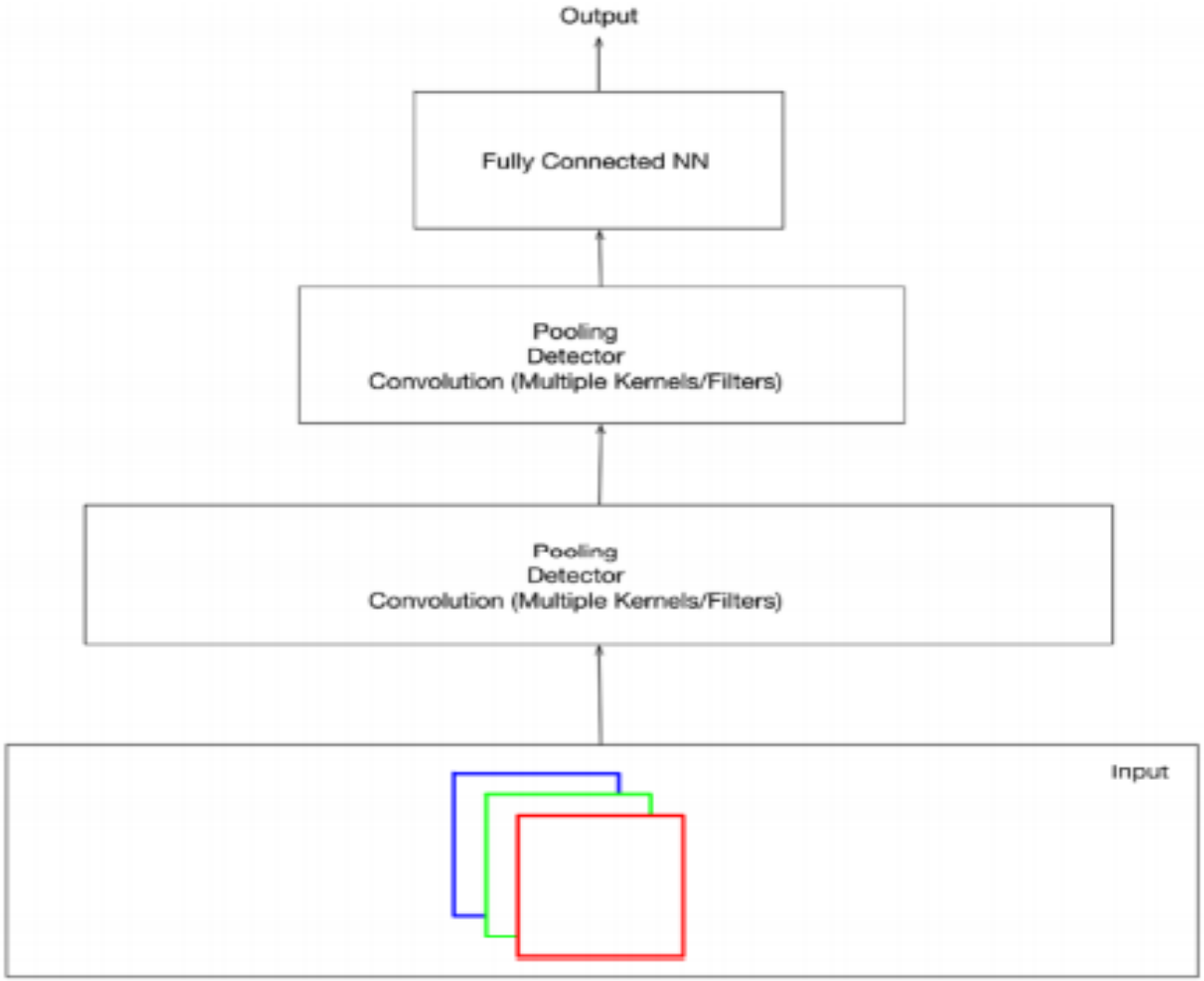
Convolution followed by detector stage and pooling



Multiple Filters/Kernels giving Multiple Feature Maps

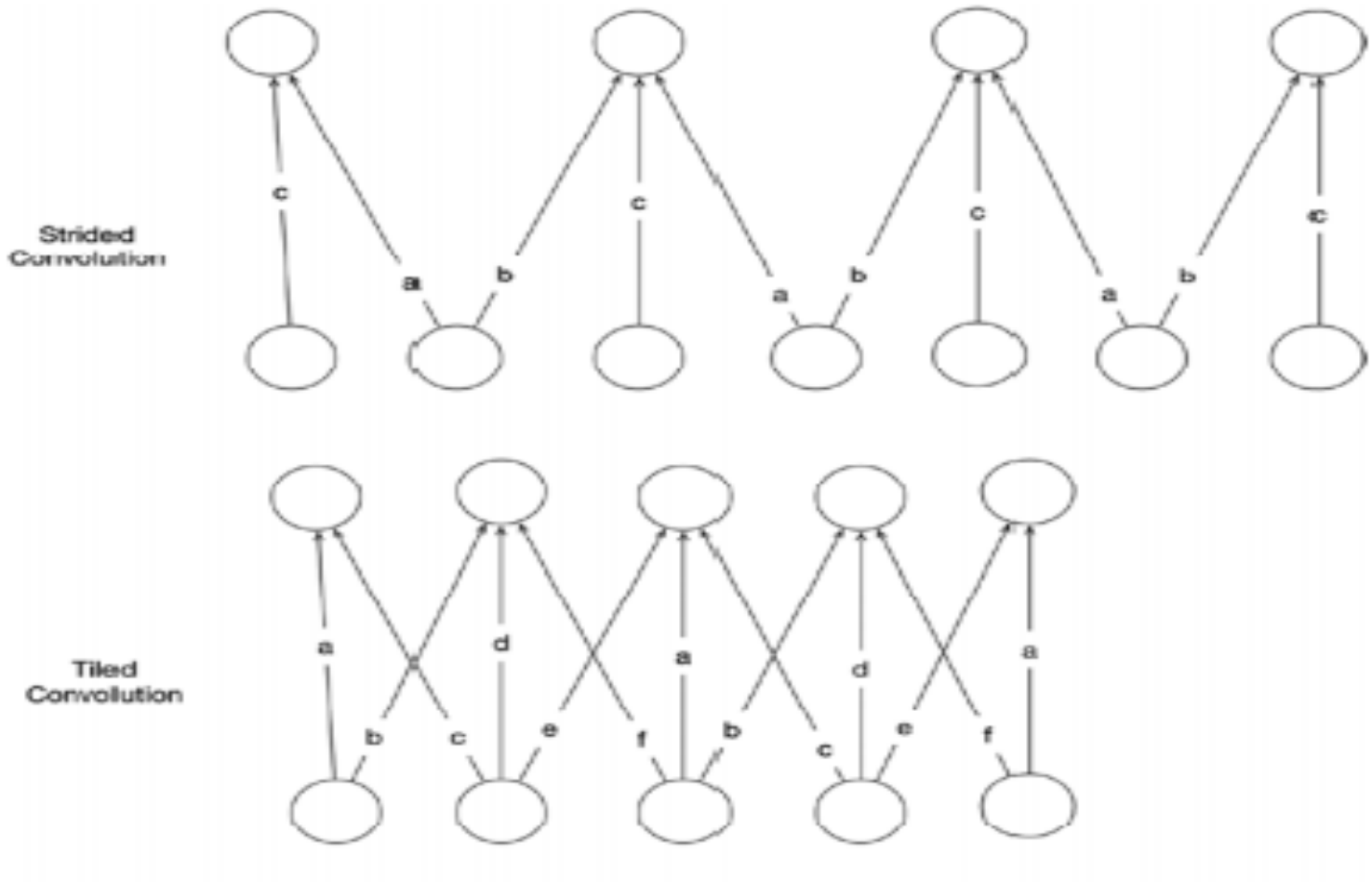


Convolution with Multiple Channels

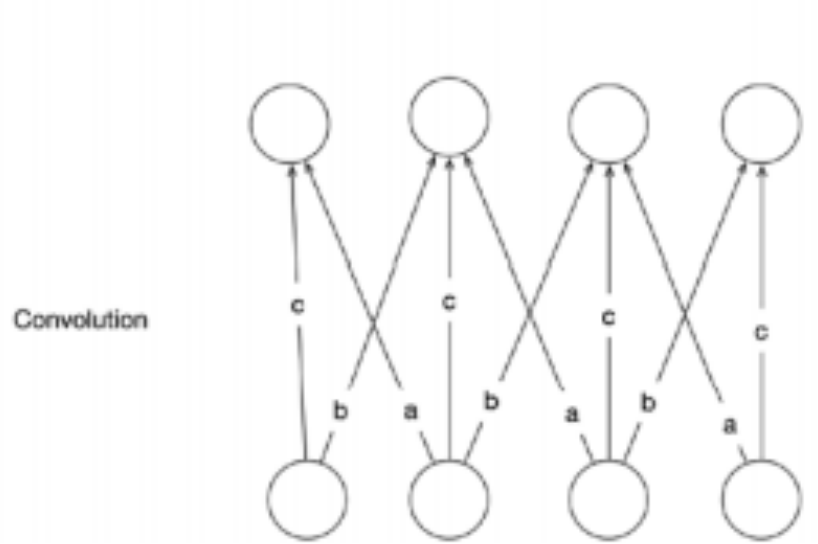


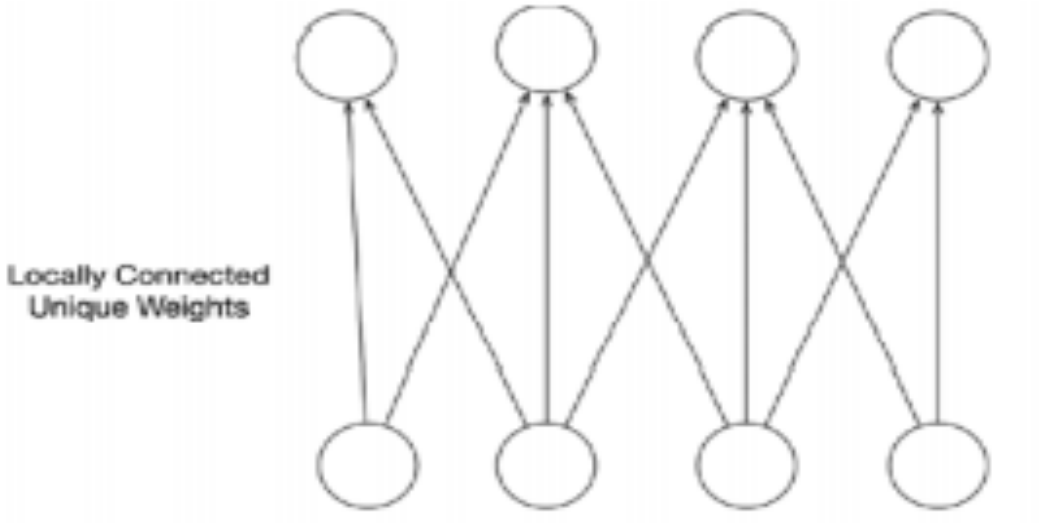
A Complete Convolution Neural Network Architecture

Convolution Variants

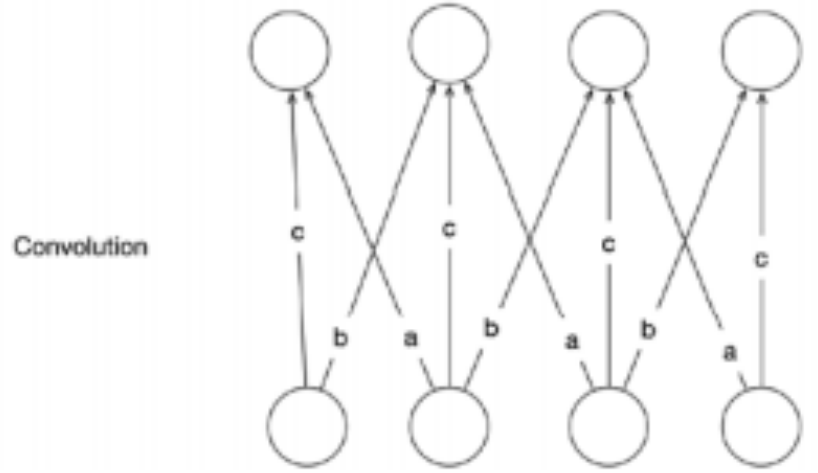


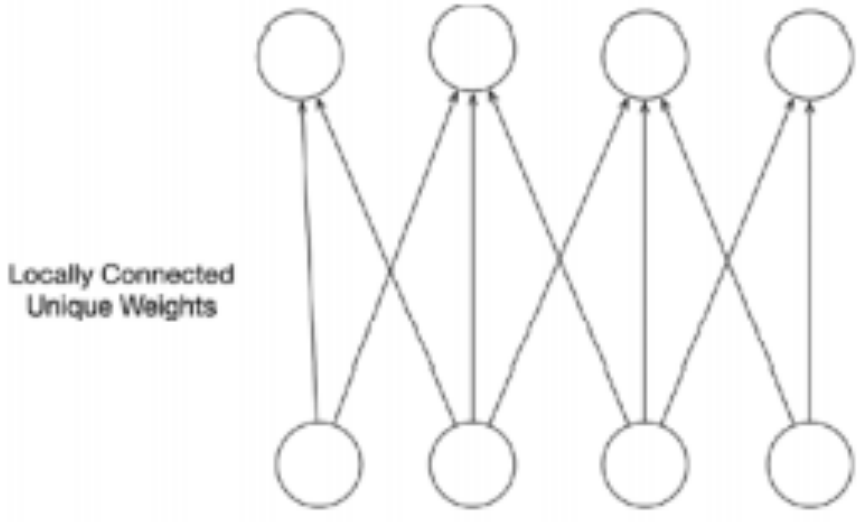
Convolution Variants





Convolution Variants





Intuition behind CNNs

- intuition behind CNNs using these building blocks.

1. The first idea to consider is the capacity of CNNs CNNs, which replace at least one of the fully connected layers of a neural network by the convolution operation, have less capacity than a fully connected network. That is, there exist data sets that a fully connected network will be able to model that a CNN will not be. **So, the first point to note is that CNNs achieve more by limiting the capacity, hence making the training efficient.**

Intuition behind CNNs

2. To consider learning the filters driving the convolution operation is, in a sense, representation learning. For instance, the learned filters might learn to detect edges, shapes, etc. The important point to consider here is that we are not manually describing the features to be extracted from the input data, **but are describing an architecture that learns to engineer the features/representations.**

3. The third idea to consider is the location invariance introduced by the pooling operation. The pooling operation separates the location of the feature from the fact that it is detected. A filter detecting straight lines might detect this filter in any portion of the image, **but the pooling operation picks the fact that the feature is detected (max pooling).**

Intuition behind CNNs

4. The fourth idea is that of hierarchy. A CNN may have multiple convolution and pooling layers stacked up followed by a fully connected network. This allows the CNN to build a hierarchy of concepts **wherein more abstract concepts are based on simpler concepts**

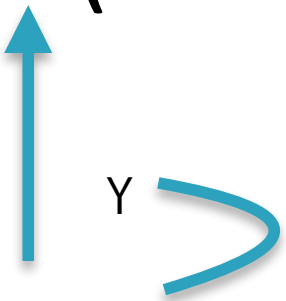
5. The fifth and last idea is the presence of a fully connected layer at the end of a series convolution and pooling layers. The idea is that the series of convolution and pooling layers **generates the features and a standard neural network learns the final classification/regression function**.

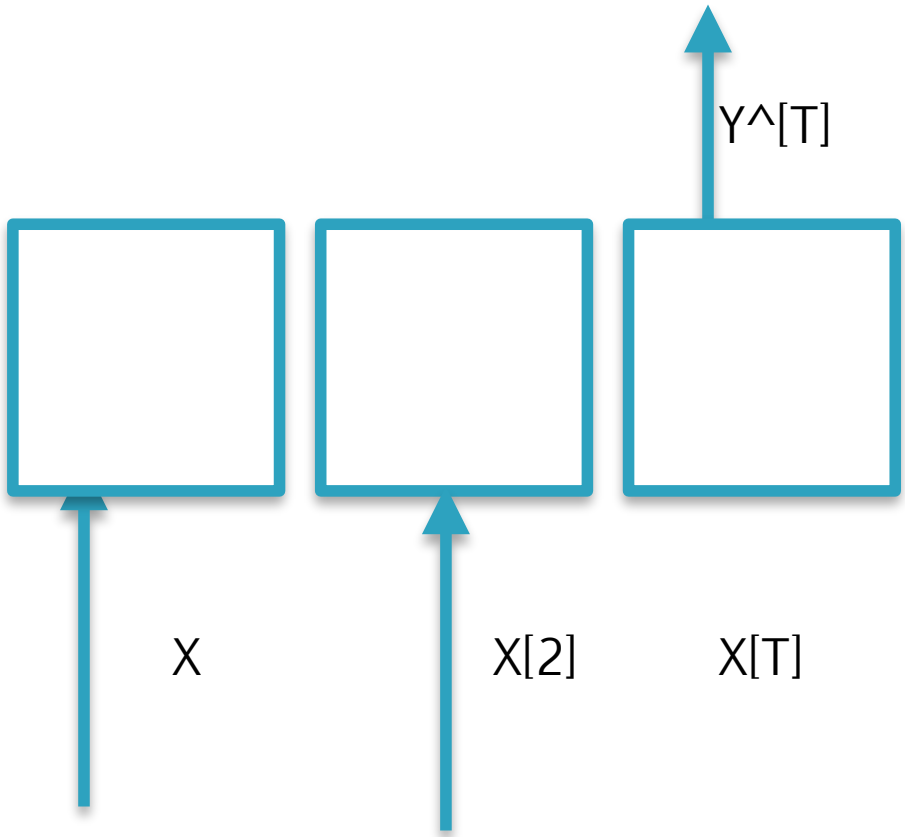
It is important to distinguish this aspect of the CNN from traditional machine learning. In traditional machine learning, an expert would hand engineer features and feed them to a neural network. In the case of CNNs, these features/representations are being learned from data.

Recurrent Neural Networks

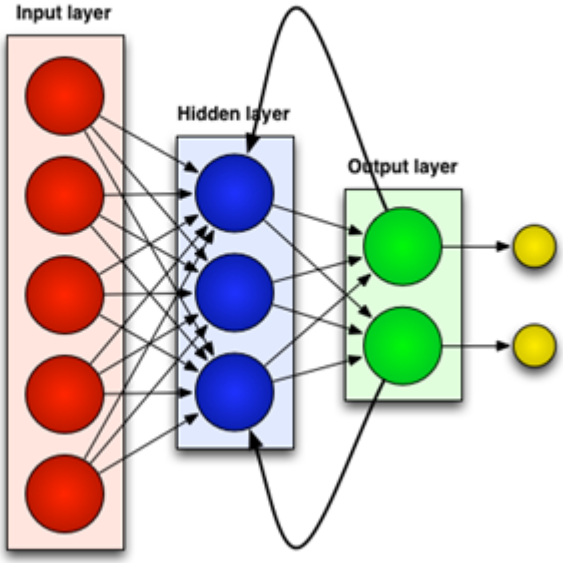
- Recurrent Neural Networks (RNNs) in essence are neural networks that employ recurrence, which is basically using information from a previous forward pass over the neural network
- RNNs are suited and have been incredibly successful when applied to problems wherein the input data on which the predictions are to be made is in the form of a sequence (series of entities where order is important).

RNN (Recurrent Neural Net)





Neural Networks That Cling to the Past



RNN BASICS

1. We will assume that input consists of a sequence of entities $x^{(1)}, x^{(2)}, \ldots, x^{(\tau)}$.
2. Corresponding to this input we either need to produce a sequence $y^{(1)}, y^{(2)}, \ldots, y^{(\tau)}$ or just one output for the entire input sequence y
3. To distinguish between what the RNN produces and what it is ideally expected to produce we will denote by $\hat{y}^{(1)}, \hat{y}^{(2)}, \ldots, \hat{y}^{(\tau)}$ or $\hat{y}$ the output the RNN produces. Note that this is distinct from what the RNN should ideally produce, which is denoted by $y^{(1)}, y^{(2)}, \ldots, y^{(\tau)}$ or y.

RNNs either produce an output for every entity in the input sequence or produce a single output for the entire sequence. Let us consider the case where an RNN produces one output for every entity in the input. The RNN can be described using the following equations:

$$h^{(t)} - \tanh\left(Ux^{(t)} + Wh^{(t-1)} + b\right)$$

$$\hat{y}^{(t)} = softmax\left(Vh^{(t)} + c\right)$$

RNN BASICS

The following points about the RNN equations should be noted:

1. The RNN computation involves first computing the hidden state for an entity in the sequence. This is denoted by $h^{(t)}$.
2. The computation of $h^{(t)}$ uses the corresponding input at entity $x^{(t)}$ and the previous hidden state $h^{(t-1)}$.
3. The output $\hat{y}^{(t)}$ is computed using the hidden state $h^{(t)}$.
4. There are weights associated with the input and the previous hidden state while computing the current hidden state. This is denoted by U and W respectively. There is also a bias term denoted by b.

RNN BASICS

5. There are weights associated with the hidden state while computing the output; this is denoted by V. There is also a bias term, which is denoted by c.
6. The tanh activation function (introduced in earlier chapters) is used in the computation of the hidden state.
7. The softmax activation function is used in the computation of the output.
8. The RNN as described by the equations can process an arbitrarily large input sequence.
9. The parameters of the RNN, namely, U,W,V,b,c, etc. are shared across the computation of the hidden layer and output value (for each of the entities in the sequence).

Figure illustrates an RNN. Note how the illustration depicts the recurrence relation with the self-loop at the hidden state.

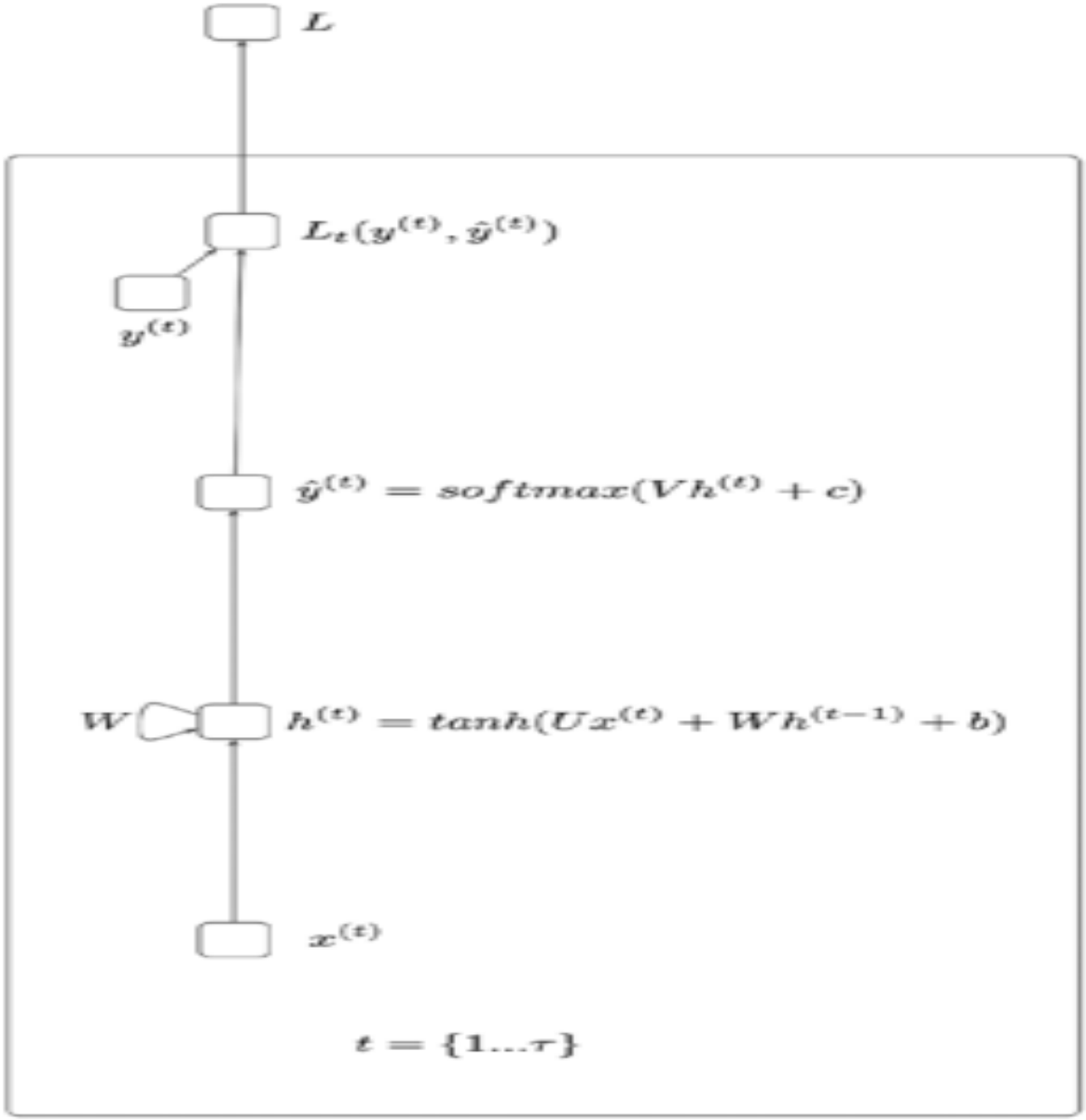
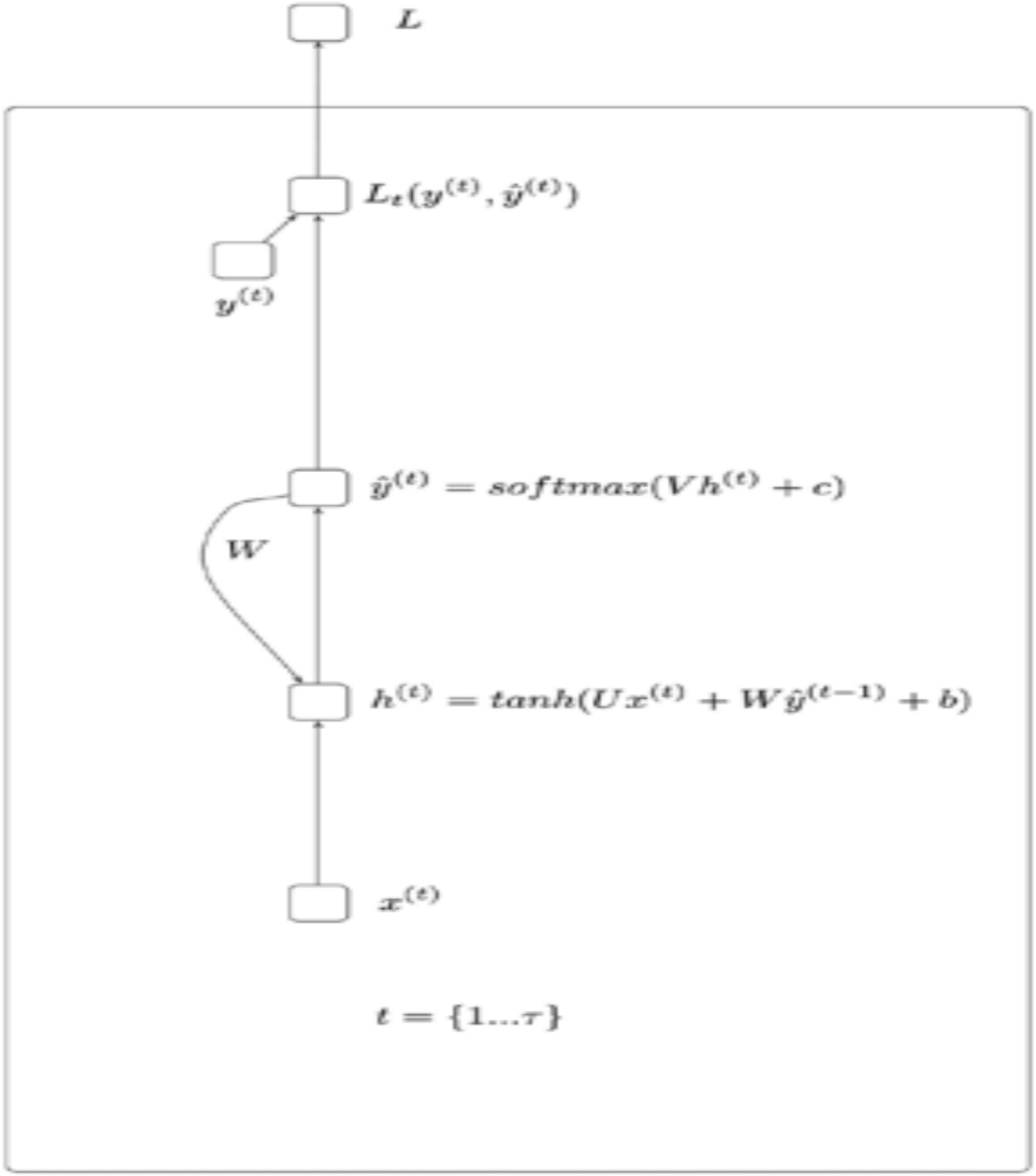


Figure 6-1. *RNN (Recurrence using the previous hida*



RNN (Recurrence using the previous output)

The equations describing such an RNN are as follows:

$$h^{(t)} = \tanh\left(Ux^{(t)} + W\hat{y}^{(t-1)} + b\right)$$

$$\hat{y}^{(t)} = softmax\left(Vh^{(t)} + c\right)$$

The following points are to be noted:

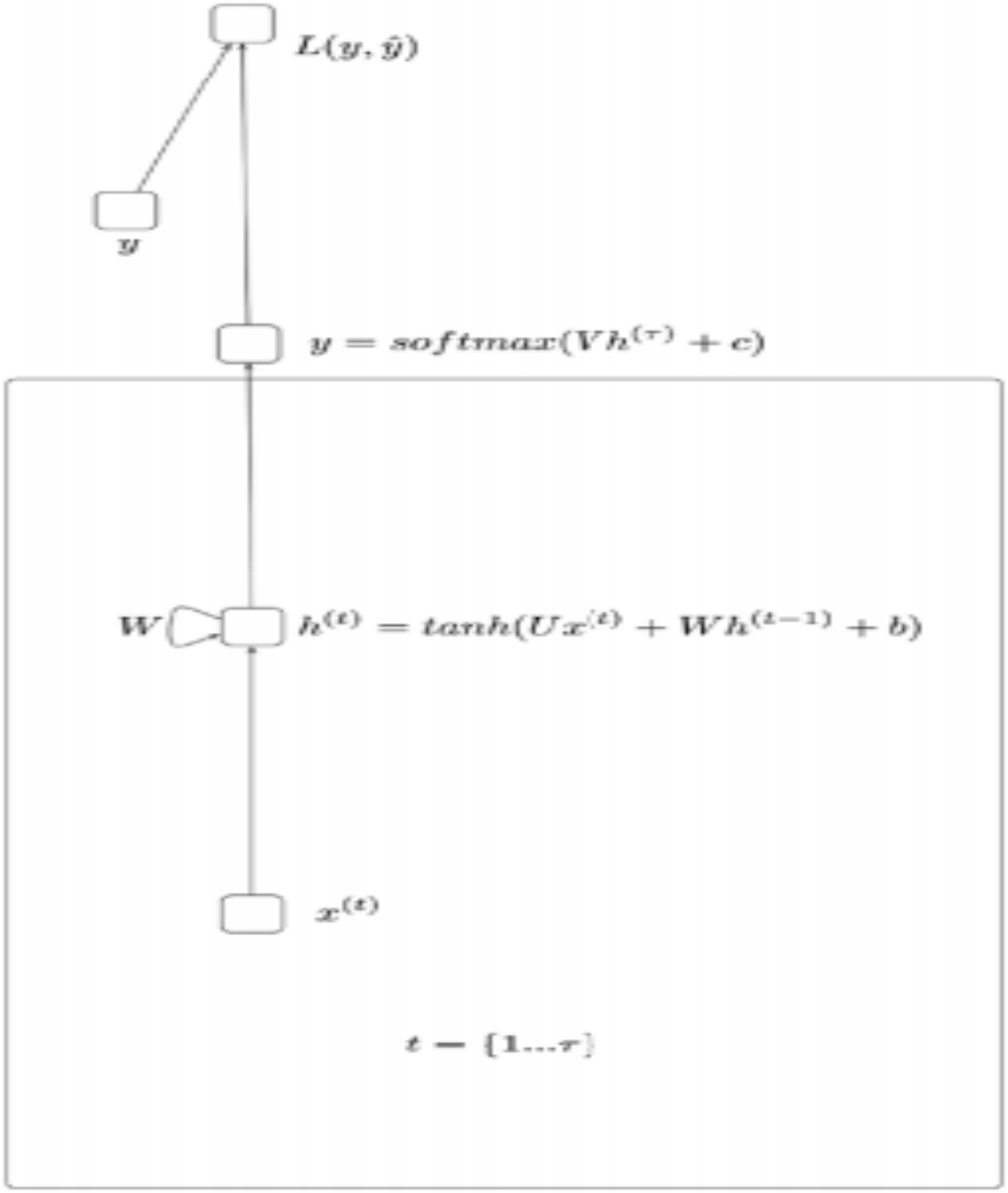
1. The RNN computation involves first computing the hidden state for an entity in the sequence. This is denoted by $h^{(t)}$.
2. The computation of $h^{(t)}$ uses the corresponding input at entity $x^{(t)}$ and the previous output $\hat{y}^{(t-1)}$.
3. The output $\hat{y}^{(t)}$ is computed using the hidden state $h^{(t)}$.
4. There are weights associated with the input and the previous output while computing the current hidden state. This is denoted by U and W respectively. There is also a bias term denoted by c.
5. There are weights associated with the hidden state while computing the output; this is denoted by V. There is also a bias term, which is denoted by c.
6. The tanh activation function (introduced in earlier chapters) is used in the computation of the hidden state.

The softmax activation function is used in the computation of the output.

Let us now consider a variation in the RNN where only a single output is produced for the entire sequence (refer to Figure 6-3). Such an RNN is described using the following equations:

$$h^{(t)} = \tanh\left(Ux^{(t)} + W\hat{y}^{(t-1)} + b\right)$$

$$\hat{y} = softmax\left(Vh^{(\tau)} + c\right)$$



RNN (Producing a single output for the entire input sequence)

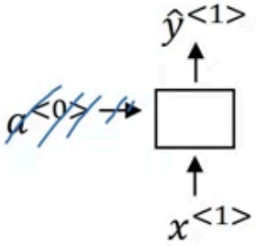
The following points are to be noted:

1. The RNN computation involves computing the hidden state for an entity in the sequence. This is denoted by $h^{(t)}$.
2. The computation of $h^{(t)}$ uses the corresponding input at entity $x^{(t)}$ and the previous hidden state $h^{(t-1)}$.
3. The computation of $h^{(t)}$ is done for each entity in the input sequence $x^{(1)}, x^{(2)}, \ldots, x^{(\tau)}$.
4. The output $\hat{y}$ is computed only using the last hidden state $h^{(\tau)}$.
5. There are weights associated with the input and the previous hidden state while computing the current hidden state. This is denoted by U and W respectively. There is also a bias term denoted by b.
6. There are weights associated with the hidden state while computing the output; this is denoted by V. There is also a bias term, which is denoted by c.
7. The tanh activation function (introduced in earlier chapters) is used in the computation of the hidden state.
8. The softmax activation function is used in the computation of the output.

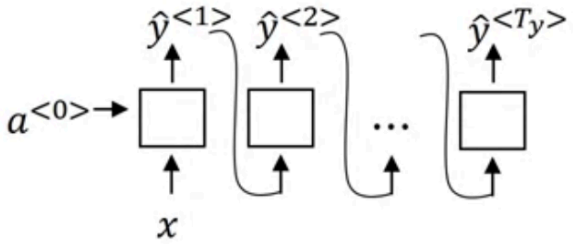
The following points are to be noted:

1. The RNN computation involves computing the hidden state for an entity in the sequence. This is denoted by $h^{(t)}$.
2. The computation of $h^{(t)}$ uses the corresponding input at entity $x^{(t)}$ and the previous hidden state $h^{(t-1)}$.
3. The computation of $h^{(t)}$ is done for each entity in the input sequence $x^{(1)}, x^{(2)}, \ldots, x^{(\tau)}$.
4. The output $\hat{y}$ is computed only using the last hidden state $h^{(\tau)}$.
5. There are weights associated with the input and the previous hidden state while computing the current hidden state. This is denoted by U and W respectively. There is also a bias term denoted by b.
6. There are weights associated with the hidden state while computing the output; this is denoted by V. There is also a bias term, which is denoted by c.
7. The tanh activation function (introduced in earlier chapters) is used in the computation of the hidden state.
8. The softmax activation function is used in the computation of the output.

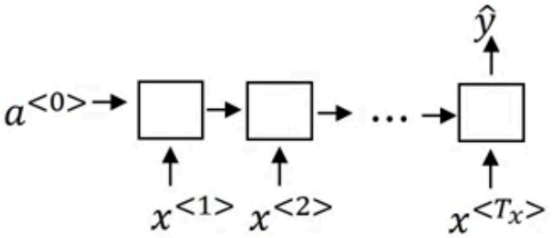
Summary of RNN types



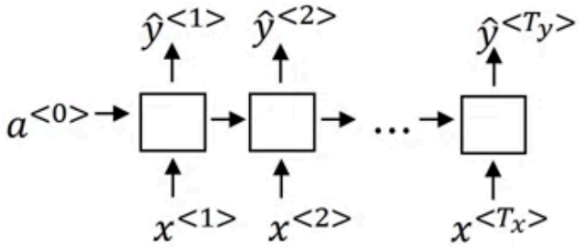
One to one



One to many

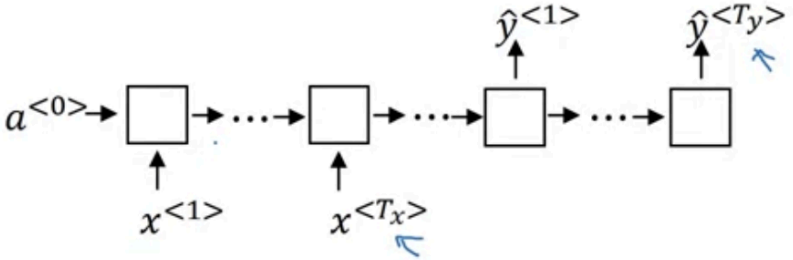


Many to one



Many to many

$T_x = T_y$



Many to many

Andrew Ng

Training RNNs:Using previous hiddden state

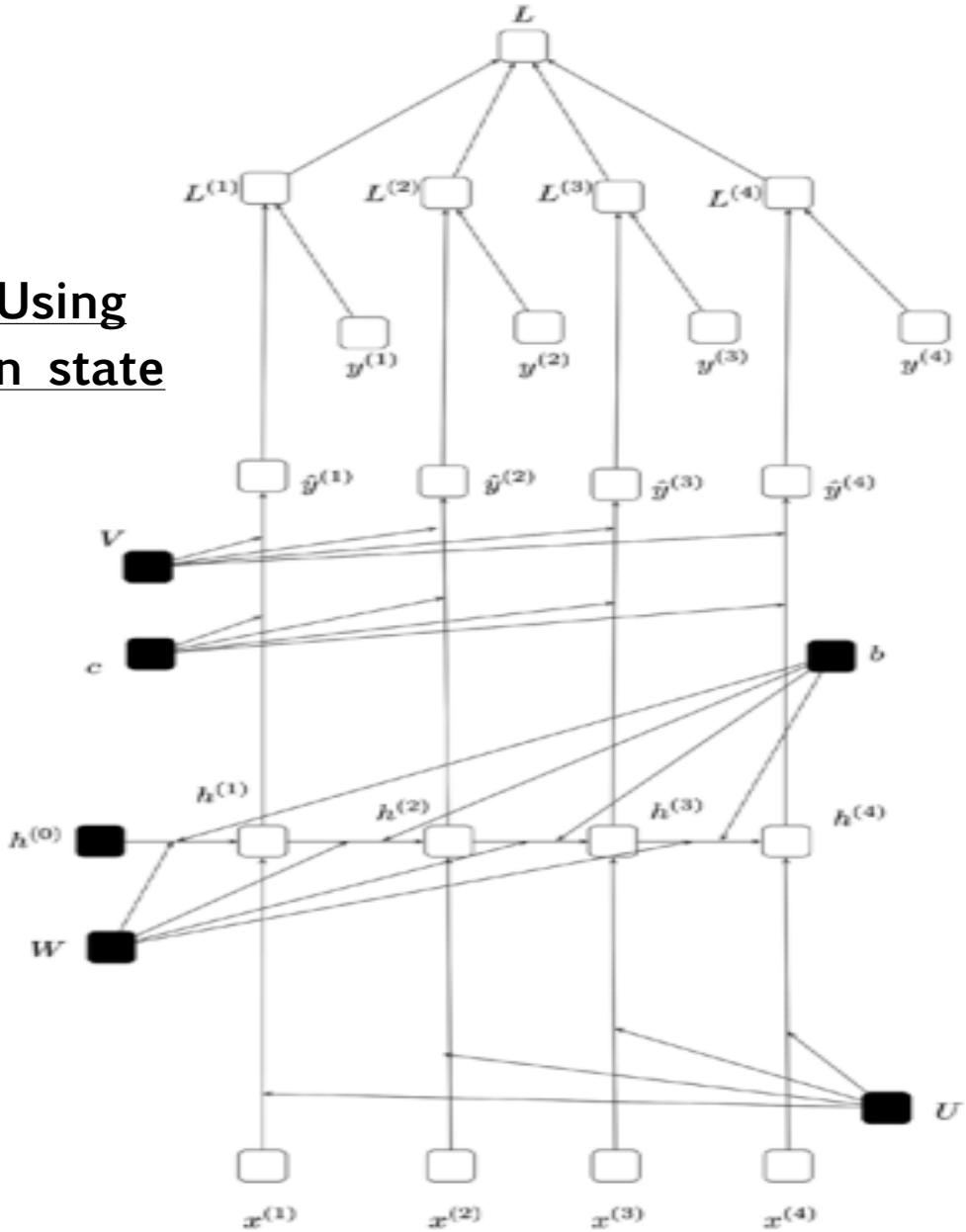


Figure 6-4. *Unrolling the RNN corresponding to Figure 6-1*

Training RNNs:Using previous outputs

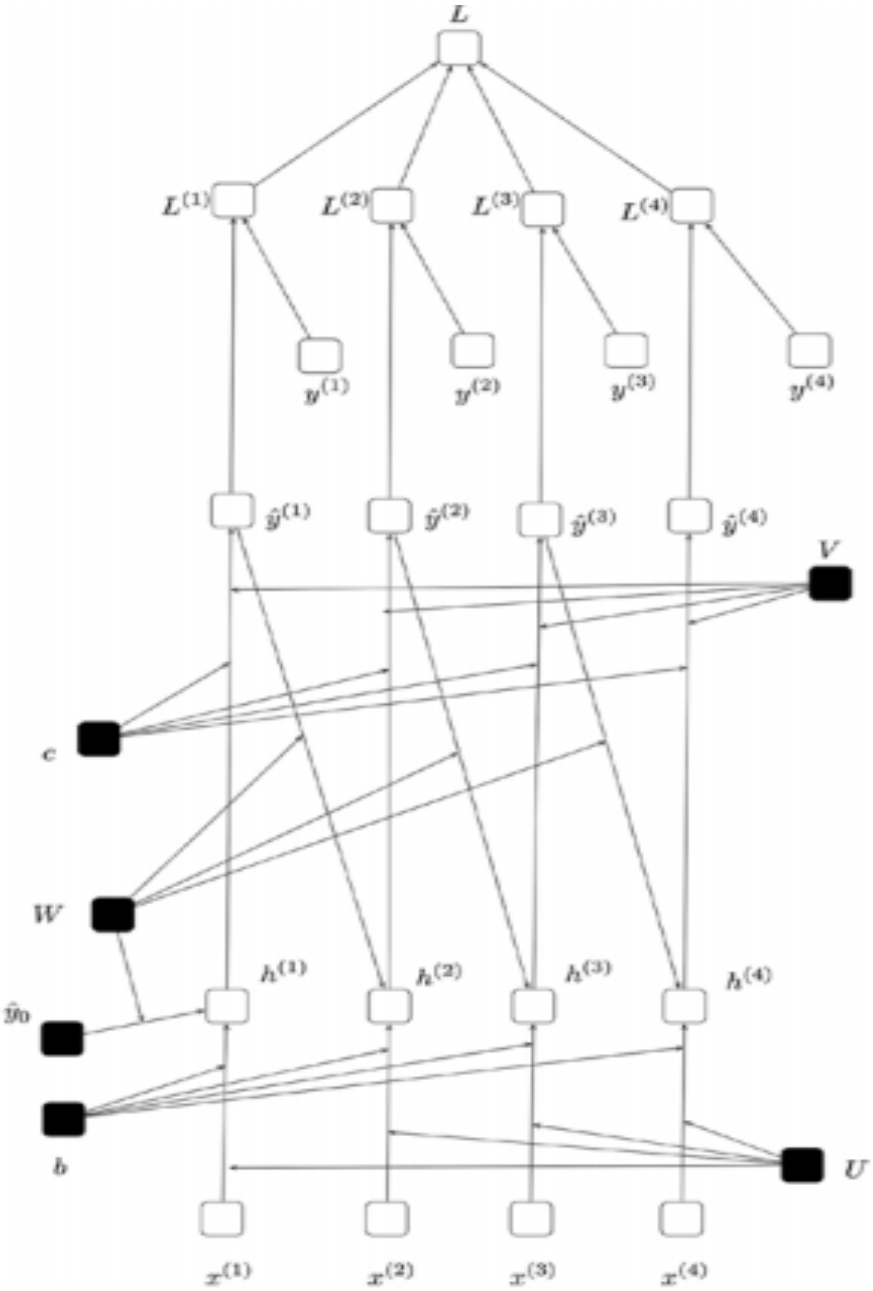


Figure 6-5. *Unrolling the RNN corresponding to Figure 6-2*

Training RNNs
single output
entire input s

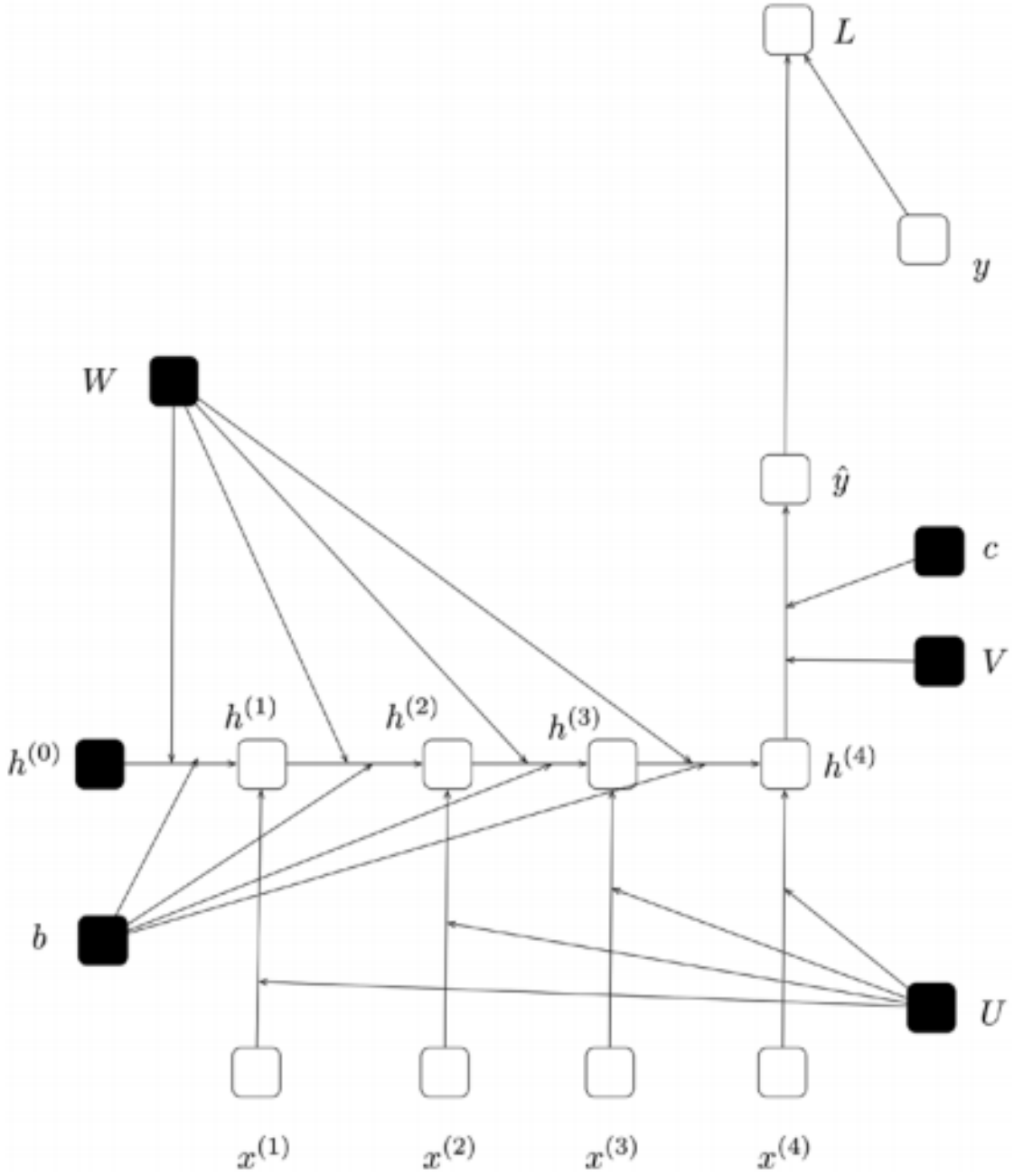


Figure 6-6. *Unrolling the RNN corresponding to Figure 6-3*

We will assume that $h^{(0)}$ is either predefined by the user, set to zero, or learned as another parameter/weight (learned like W, V, or b). Unrolling simply means writing out the equations describing the RNN in terms of $h^{(0)}$. Of course, in order to do so, we need fix the length of the sequence, which is denoted by τ. Figure 6-4 illustrates the unrolled RNN corresponding to the RNN in Figure 6-1 assuming an input sequence of size 4. Similarly, Figure 6-5 and 6-6 illustrate the unrolled RNNs corresponding to the RNNs in Figure 6-2 and 6-3 respectively. The following points are to be noted:

1. The unrolling process operates on the assumption that the length of the input sequence is known beforehand and based on the recurrence is unrolled.

2. Once unrolled, we essentially have a non-recurrent neural network.

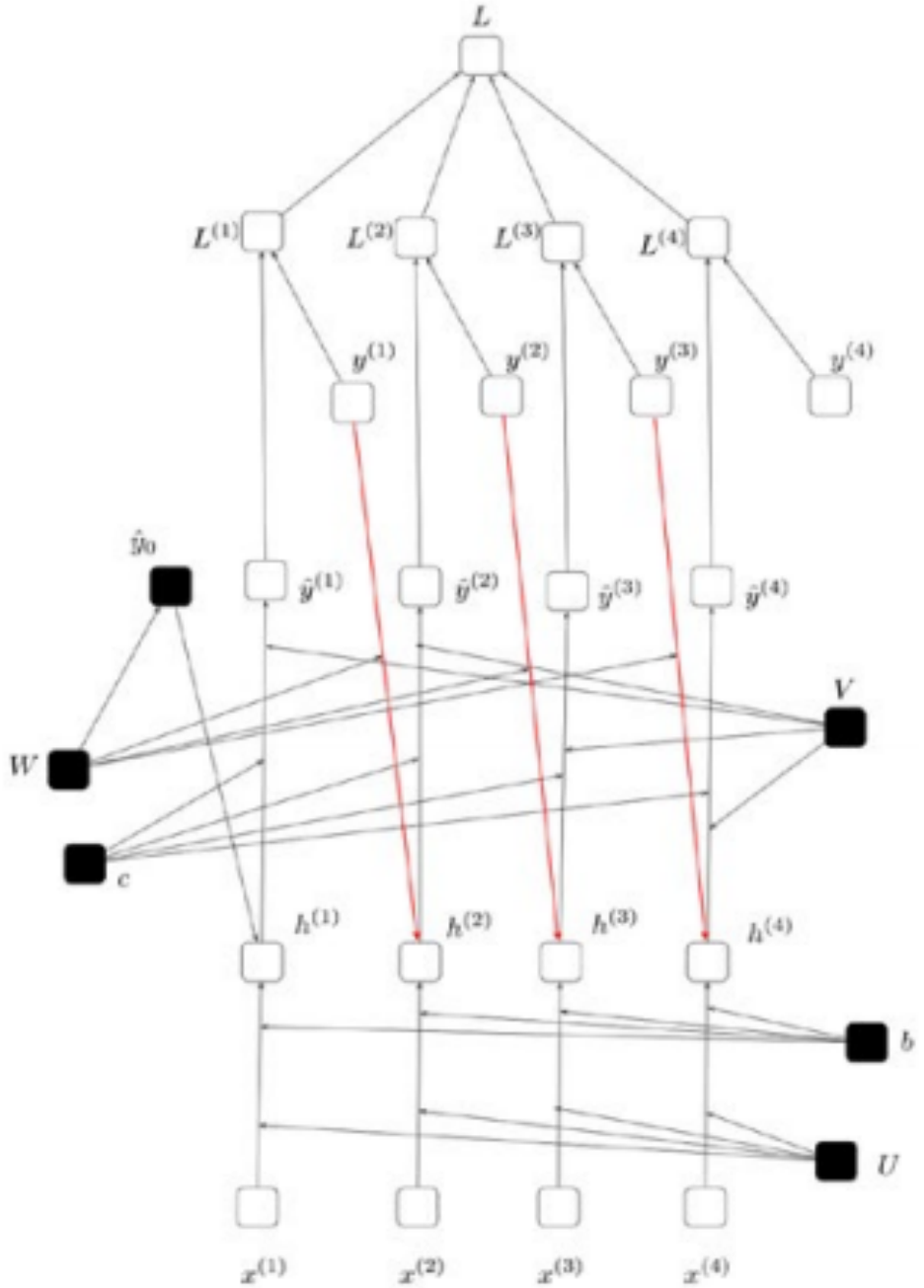
3. The parameters to be learned, namely, U,W,V,b,c, etc. (denoted in dark in the diagram) are shared across the computation of the hidden layer and output value. We have seen such parameter sharing earlier in the context of CNNs.

4. Given an input and output of a given size, say τ (assumed to be 4 in Figures 6-4, 6-5, 6-6), we can unroll an RNN and compute gradients for the parameters to be learned with respect to a loss function (as we have seen in earlier chapters).

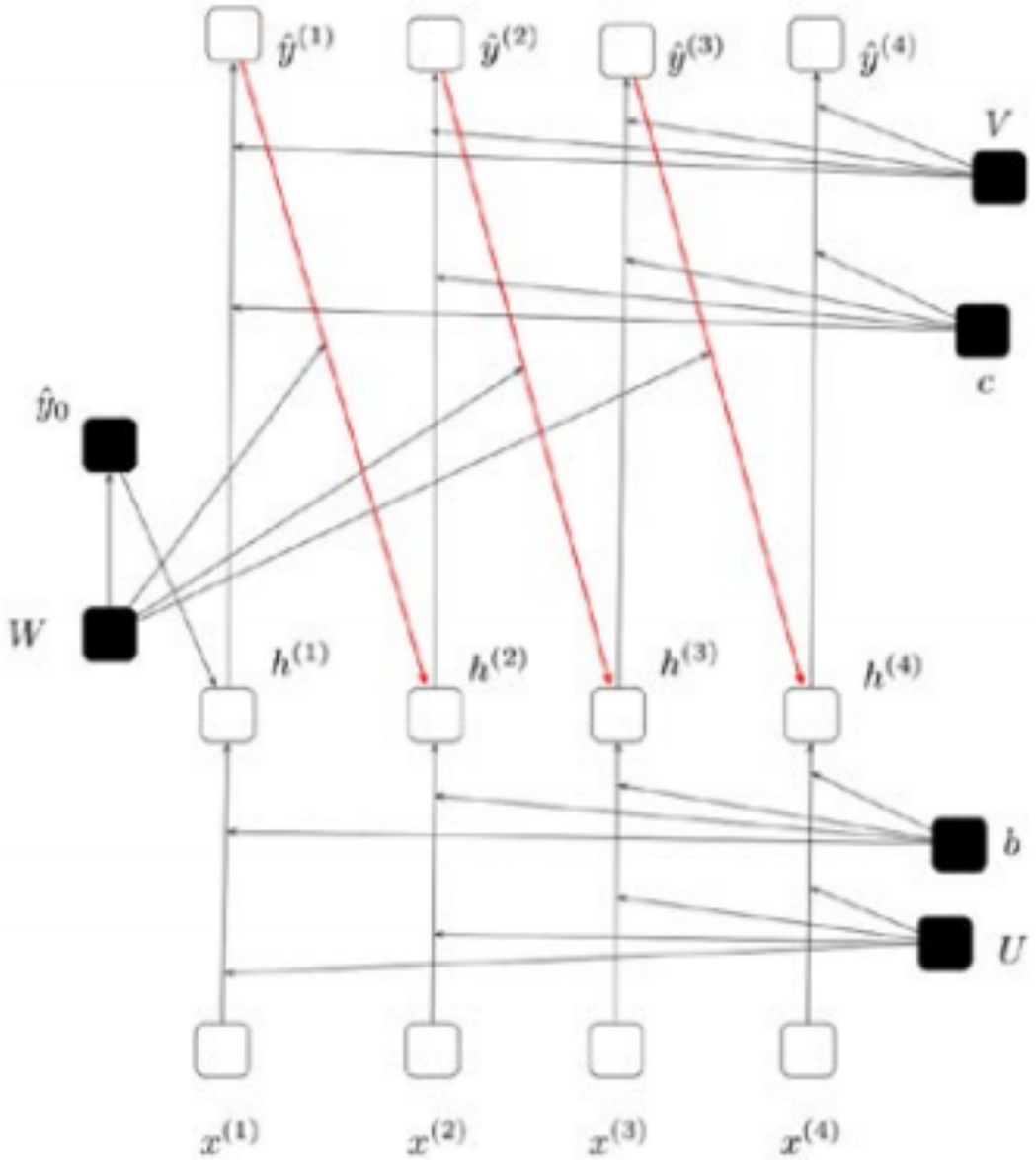
5. Thus, training an RNN is simply unrolling the RNN for a given size of input (and, correspondingly, the expected output) and training the unrolled RNN via computing the gradients and using stochastic gradient descent.

- Unrolling the RNN corresponding to Figure 6-1 for different sizes of inputs
- Given that the data set to be trained on consists of sequences of varying sizes, the input sequences are grouped so that the sequences of the same size fall in one group. Then for a group, we can unroll the RNN for the sequence length and train. Training for a different group will require the RNN to be unrolled for a different sequence length. Thus, it is possible to train the RNN on inputs of varying sizes by unrolling and training with the unrolling done based on the sequence length.

Teacher Forcing (Training)



Teacher Forcing (Prediction)



Bidirectional RNNs

- The key intuition behind **a bidirectional RNN is to use the entities that lie further in the sequence to make a prediction for the current entity.** For all the RNNs we have considered so far we have been using the previous entities (captured by the hidden state) and the current entity in the sequence to make the prediction. However, we have not been using information concerning the entities that lie further in the sequence to make predictions.
- A bidirectional RNN leverages this information and can give improved predictive accuracy in many cases

Bidirectional RNNs

$$h_f^{(t)} = \tanh\left(U_f x^{(t)} + W_f h^{(t+1)} + b_f\right)$$

$$h_b^{(t)} = \tanh\left(U_b x^{(t)} + W_b h^{(t-1)} + b_b\right)$$

$$\hat{y}^{(t)} = softmax\left(V_b h_b^{(t)} + V_f h_f^{(t)} + c\right)$$

The following points are to be noted:

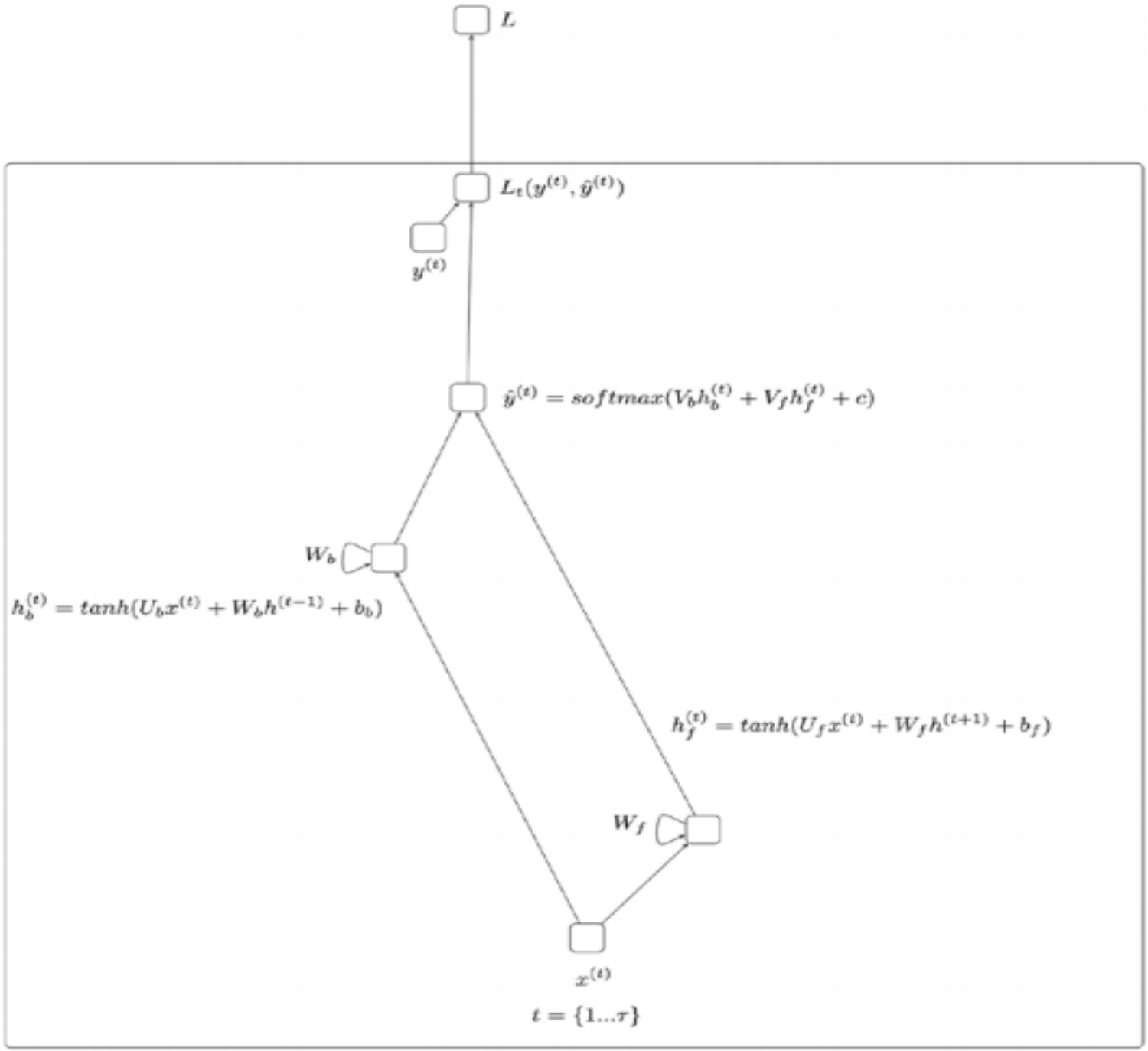
1. The RNN computation involves first computing the forward hidden state and backward hidden state for an entity in the sequence. This is denoted by $h_f^{(t)}$ and $h_b^{(t)}$ respectively.
2. The computation of $h_f^{(t)}$ uses the corresponding input at entity $x^{(t)}$ and the previous hidden state $h_f^{(t-1)}$.
3. The computation of $h_b^{(t)}$ uses the corresponding input at entity $x^{(t)}$ and the previous hidden state $h_b^{(t-1)}$.

Bidirectional RNNs

4. The output $\hat{y}^{(t)}$ is computed using the hidden state $h_f^{(t)}$ and $h_b^{(t)}$
5. There are weights associated with the input and the previous hidden state while computing the current hidden state. This is denoted by U_f, W_f, U_b, and W_b respectively. There is also a bias term denoted by b_f and b_b.
6. There are weights associated with the hidden state while computing the output; this is denoted by V_b and V_f. There is also a bias term, which is denoted by c.
7. The tanh activation function (introduced in earlier chapters) is used in the computation of the hidden state.
8. The softmax activation function is used in the computation of the output.
9. The RNN as described by the equations can process an arbitrarily large input sequence.
10. The parameters of the RNN, namely, U_f, U_b, W_f, W_b, V_b, V_f, b_f, b_b, c, etc. are shared across the computation of the hidden layer and output value (for each of the entities in the sequence).

Bidirectional RNNs

4. The output $\hat{y}^{(t)}$ is computed using the hidden state $h_f^{(t)}$ and $h_b^{(t)}$
5. There are weights associated with the input and the previous hidden state while computing the current hidden state. This is denoted by U_f, W_f, U_b, and W_b respectively. There is also a bias term denoted by b_f and b_b.
6. There are weights associated with the hidden state while computing the output; this is denoted by V_b and V_f. There is also a bias term, which is denoted by c.
7. The tanh activation function (introduced in earlier chapters) is used in the computation of the hidden state.
8. The softmax activation function is used in the computation of the output.
9. The RNN as described by the equations can process an arbitrarily large input sequence.
10. The parameters of the RNN, namely, U_f, U_b, W_f, W_b, V_b, V_f, b_f, b_b, c, etc. are shared across the computation of the hidden layer and output value (for each of the entities in the sequence).


L
$L_t(y^{(t)}, \hat{y}^{(t)})$
$y^{(t)}$
$\hat{y}^{(t)} = softmax(V_b h_b^{(t)} + V_f h_f^{(t)} + c)$
W_b
$h_b^{(t)} = tanh(U_b x^{(t)} + W_b h^{(t-1)} + b_b)$
$h_f^{(t)} = tanh(U_f x^{(t)} + W_f h^{(t+1)} + b_f)$
W_f
$x^{(t)}$
$t = \{1...\tau\}$

Gradient Explosion and Vanishing

- Training RNNs suffers from the challenges of vanishing and explosion of gradients. Vanishing gradients means that, when computing the gradients on the unrolled RNNs, the value of the gradients can drop to a very small value (close to zero).
- Similarly, the gradients can increase to a very high value which is referred to as the exploding gradient problem. In both cases, training the RNN is a challenge.

Let us relook at the equations describing the RNN.

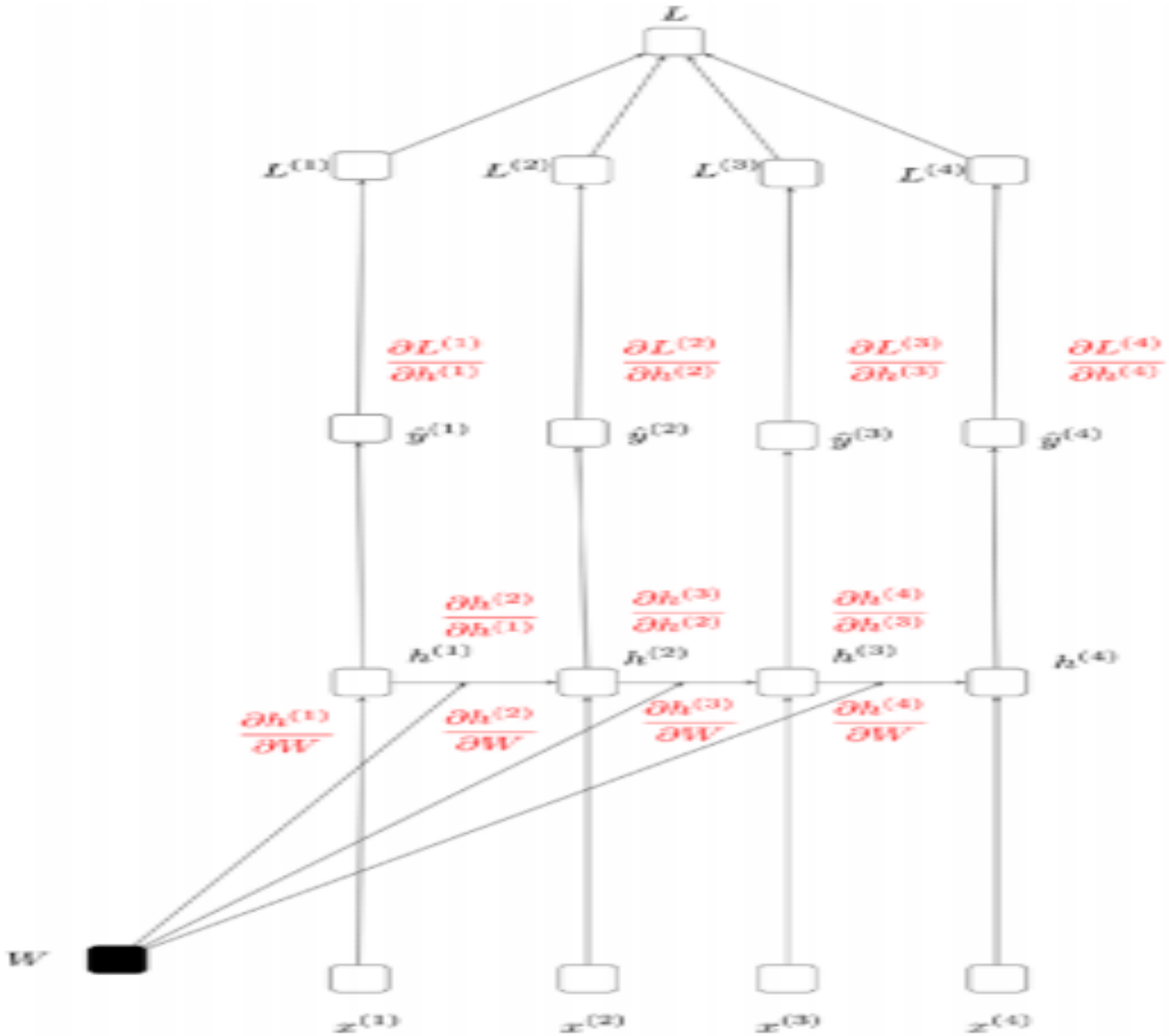
$$h^{(t)} = \tanh\left(Ux^{(t)} + Wh^{(t-1)} + b\right)$$

$$\hat{y}^{(t)} = softmax\left(Vh^{(t)} + c\right)$$

Let us derive the expression for the $\frac{\partial L}{\partial W}$ by applying the chain rule. This is illustrated in Figure 6-10.

$$\frac{\partial L}{\partial W} = \sum_{1 \le t \le \tau} \frac{\partial L^{(t)}}{\partial h^{(t)}} \left[\sum_{1 \le k \le t} \left[\prod_{k \le j \le t-1} \frac{\partial h^{(j+1)}}{\partial h^{(j)}} \right] \frac{\partial h^{(k)}}{\partial W} \right]$$

Let us now focus on the part of the expression $\prod_{k \le j \le t-1} \frac{\partial h^{(j+1)}}{\partial h^{(j)}}$ which involves a repeated matrix multiplication of W which contributes to both the vanishing and exploding gradient problems. Intuitively, this is similar to multiplying a real valued number over and over again, which might lead to the product shrinking to zero or exploding to infinity.



$$\frac{\partial L}{\partial W} = \sum_{1 \le t \le \tau} \frac{\partial L^{(t)}}{\partial h^{(t)}} \left[\sum_{1 \le k \le t} \left[\prod_{k \le j \le t-1} \frac{\partial h^{(j+1)}}{\partial h^{(j)}} \right] \frac{\partial h^{(k)}}{\partial W} \right]$$

Gradient Clipping

One simple technique to deal with exploding gradients is to rescale the norm of gradient whenever it goes over a user-defined threshold. Specifically, if the gradient denoted by $\hat{g} = \frac{\partial L}{\partial W}$ and if $\left|\hat{g}\right| > c$ then we set $\hat{g} = \frac{c}{\left|\hat{g}\right|}\hat{g}$.

This technique is both simple and computationally efficient but does introduce an extra hyperparameter.

Long Short Term Memory

Let us now take a look at another variation on RNNs, namely, the Long Short Term Memory (LSTM) Network. An LSTM can be described with the following set of equations. Note that the symbol $\odot$. notes pointwise multiplication of two vectors (if $a=[1,1,2]$ and $b=[0.5,0.5,0.5]$, then $a \odot b=[0.5,0.5,1]$, the functions σ, g and h are non-linear activation functions, all W and R are weight matrices, and all the b terms are bias terms).

$$z^{(t)} = g\left(W_z x^{(t)} + R_z \ \hat{y}^{(t-1)} + b_z\right)$$

$$i^{(t)} = \sigma\left(W_i x^{(t)} + R_i \ \hat{y}^{(t-1)} + p_i \odot c^{(t-1)} + b_i\right)$$

$$f^{(t)} = \sigma\left(W_f x^{t} + R_f \ \hat{y}^{(t-1)} + p_f \odot c^{(t-1)} + b_f\right)$$

$$c^{(t)} = i^{(t)} \odot z^{(t)} + f^{(t)} \odot c^{(t-1)}$$

$$o^{(t)} = \sigma\left(W_o x^{(t)} + R_o \ \hat{y}^{(t-1)} + p_o \odot c^{(t)} + b_o\right)$$

Long Short Term Memory

Let us now take a look at another variation on RNNs, namely, the Long Short Term Memory (LSTM) Network. An LSTM can be described with the following set of equations. Note that the symbol $\odot$ notes pointwise multiplication of two vectors (if $a=[1,1,2]$ and $b=[0.5,0.5,0.5]$, then $a \odot b=[0.5,0.5,1]$, the functions σ, g and h are non-linear activation functions, all W and R are weight matrices, and all the b terms are bias terms).

$$z^{(t)} = g\left(W_z x^{(t)} + R_z \hat{y}^{(t-1)} + b_z\right)$$

$$i^{(t)} = \sigma\left(W_i x^{(t)} + R_i \hat{y}^{(t-1)} + p_i \odot c^{(t-1)} + b_i\right)$$

$$f^{(t)} = \sigma\left(W_f x^{t} + R_f \hat{y}^{(t-1)} + p_f \odot c^{(t-1)} + b_f\right)$$

$$c^{(t)} = i^{(t)} \odot z^{(t)} + f^{(t)} \odot c^{(t-1)}$$

$$o^{(t)} = \sigma\left(W_o x^{(t)} + R_o \hat{y}^{(t-1)} + p_o \odot c^{(t)} + b_o\right)$$

The following points are to be noted:

1. The most important element of the LSTM is the cell state denoted by $c^{(t)} = i^{(t)} \odot z^{(t)} + f^{(t)} \odot c^{(t-1)}$. The cell state is updated based on the block input $z^{(t)}$ and the previous cell state $c^{(t-1)}$. The input gate $i^{(t)}$ determines what fraction of the block input makes it into the cell state (hence called a gate). The forget gate $f^{(t)}$ determines how much of the previous cell state to retain.

2. The output $\hat{y}^{(t)}$ is determined based on the cell state $c^{(t)}$ and the output gate $o^{(t)}$, which determines how much the cell state affects the output.

3. The $z^{(t)}$ term is referred to as the block input and it produces a value based on the current input and the previous output.

4. The $i^{(t)}$ term is referred to as the input gate. It determines how much of the input to retain in the cell state $c^{(t)}$.

5. All the p terms are peephole connections, which allow for a faction of the cell state to factor into the computation of the term in question.

6. The computation of the cell state $c^{(t)}$ does not encounter the issue of the vanishing gradient (this is referred to as the constant error carousal). However, LSTMs are affected by exploding gradients and gradient clipping is used while training.

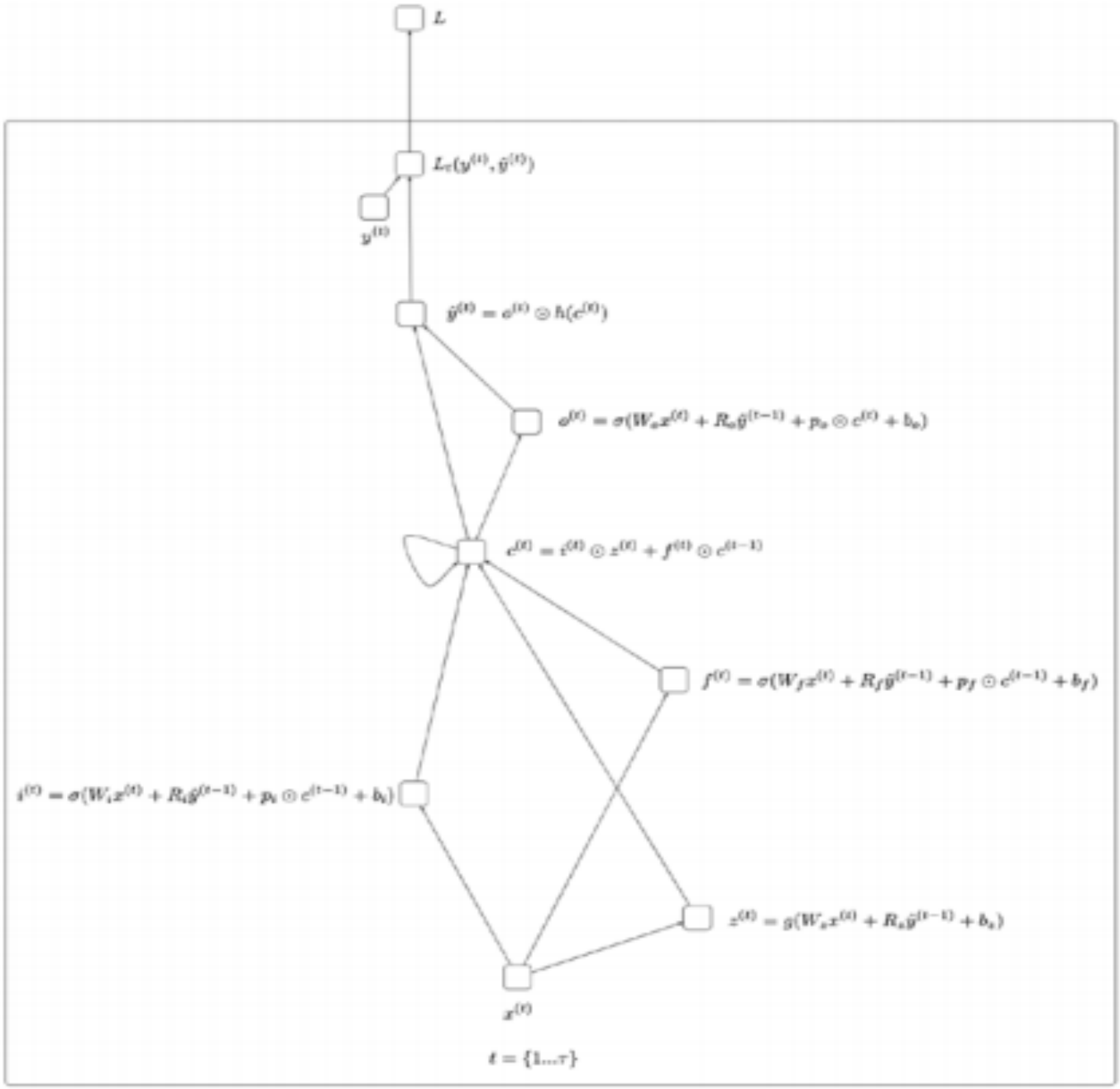


Figure 6-11. *Long Short Term Memory*

Summary

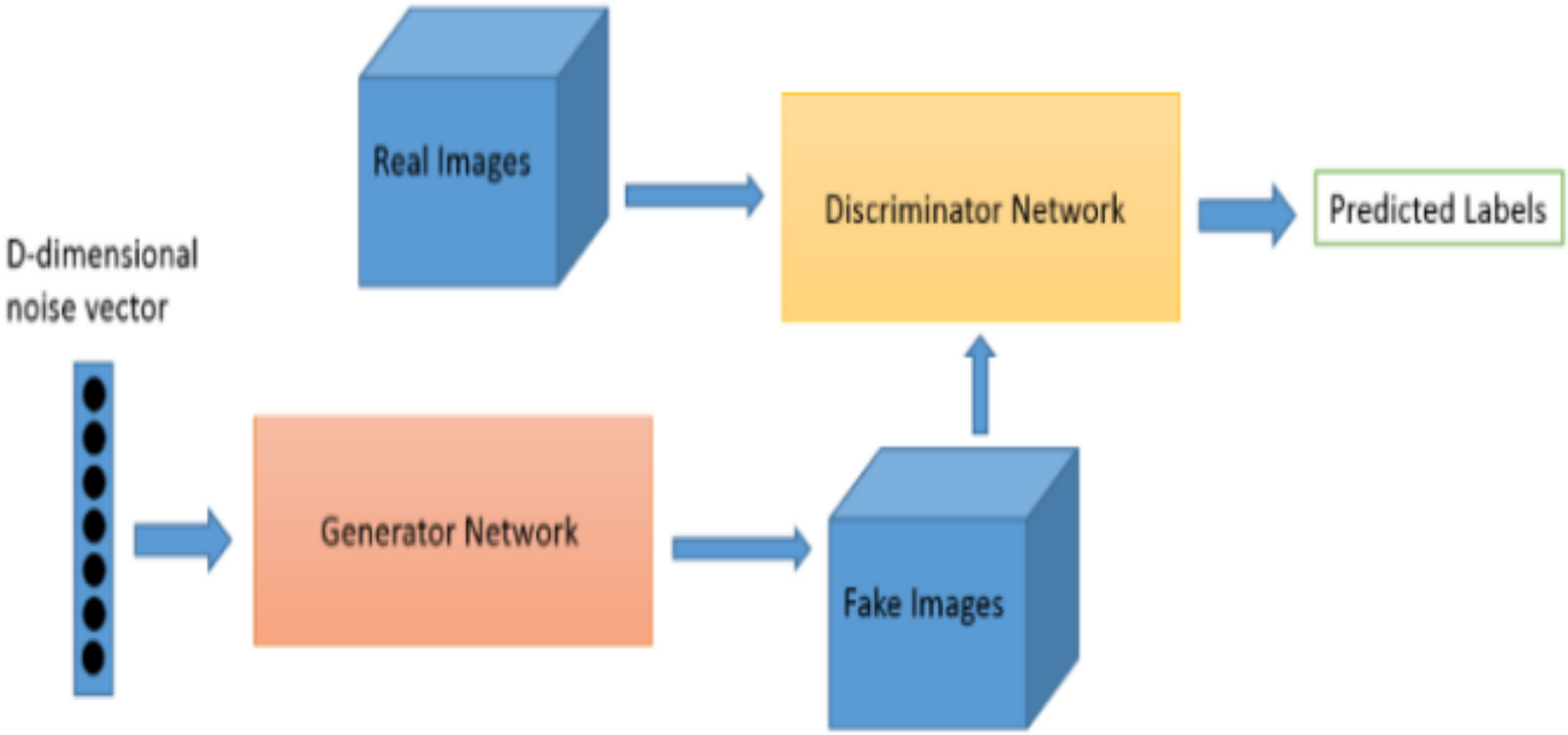
- Training RNNs via unrolling (backpropagation through time), the problem of vanishing and exploding gradients, and long short term memory networks. It is important to internalize how RNNs contain internal/hidden states that allow them to make predictions on a sequence of inputs, an ability that goes beyond conventional neural networks.

Generative Adversarial Networks (GAN)

- GANs were invented by Ian Goodfellow, who's now staff research scientist at Google Brain, and his associates from the University of Montreal in 2014. Yann LeCun, the director of Facebook AI said: "Generative Adversarial Networks is the most interesting idea in the last ten years in Machine Learning." GAN makes the neural nets more human by allowing it to CREATE rather than just training it with data sets.

Generative Adversarial Networks (GAN)

- A generative adversarial network is composed of two neural networks: a **generative** network and a **discriminative** network. In the starting phase, a Generator model takes random noise signals as input and generates a random noisy (fake) image as the output. Gradually with the help of the Discriminator, it starts generating images of a particular class that look real.



Credit: O'Reilly

Source: O'Reilly. Generator and Discriminator are pitting one against the other (thus the "adversarial") and compete during the training where their losses push against each other to improve behaviors (via backpropagation). The goal of the generator is to pass without being caught while the goal of the discriminator is to identify the fakes.

Natural Language Processing (NLP)

- Actually NLP is a broader topic though it gained huge popularity recently thanks to machine learning. NLP is the ability of computers to analyze, understand and generate human language, including speech. For example you can do sentiment analysis given any text. NLP can make AI recommendations after parsing thru movie/book reviews or web.
- NLP can run chatbots/digital assistants for front end tasks using text or audio interactions. Alexa/ Siri/Cortana/Google Assistant are the famous digital personas using NLP engines.

The Artificial Intelligence (AI) behind Chat bots

www.bluelimelabs.com

THANK YOU