Neural Networks and Deep Learning with R & Python

Part Two: 23July24

Data Science Group

Mohammad Arashi

linkedin.com/in/mohammadarashi

Professor and Director of Data Science Laboratory, Ferdowsi University of Mashhad, Iran
Extraordinary Professor, University of Pretoria, South Africa

Reminder to myself

We learn about CNN and RNN

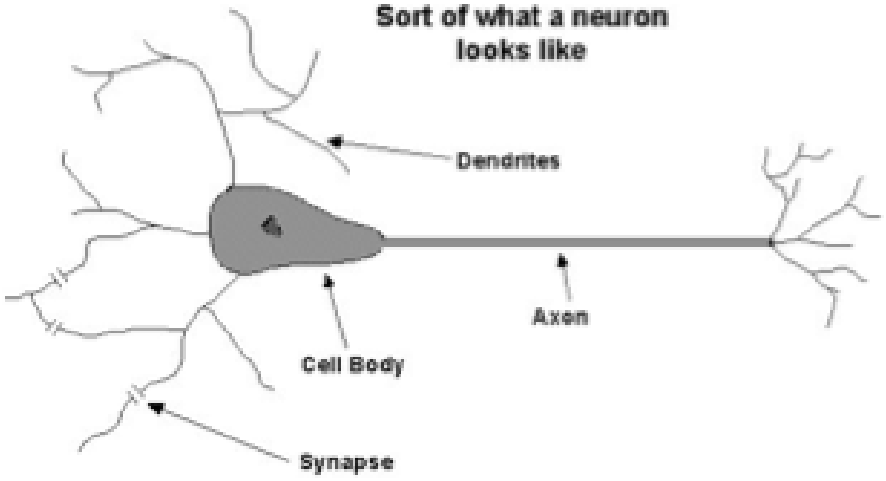
GitHub-https://github.com/M-Arashi/SASA-DS

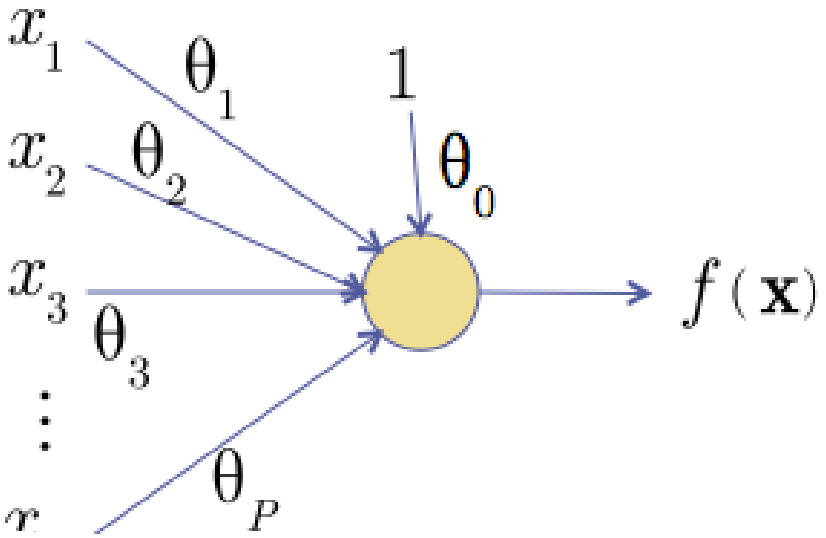
Install Anaconda

https://www.anaconda.com/download

Posit-Google Colab

Tom Yeh-linkedin.com/in/tom-yeh

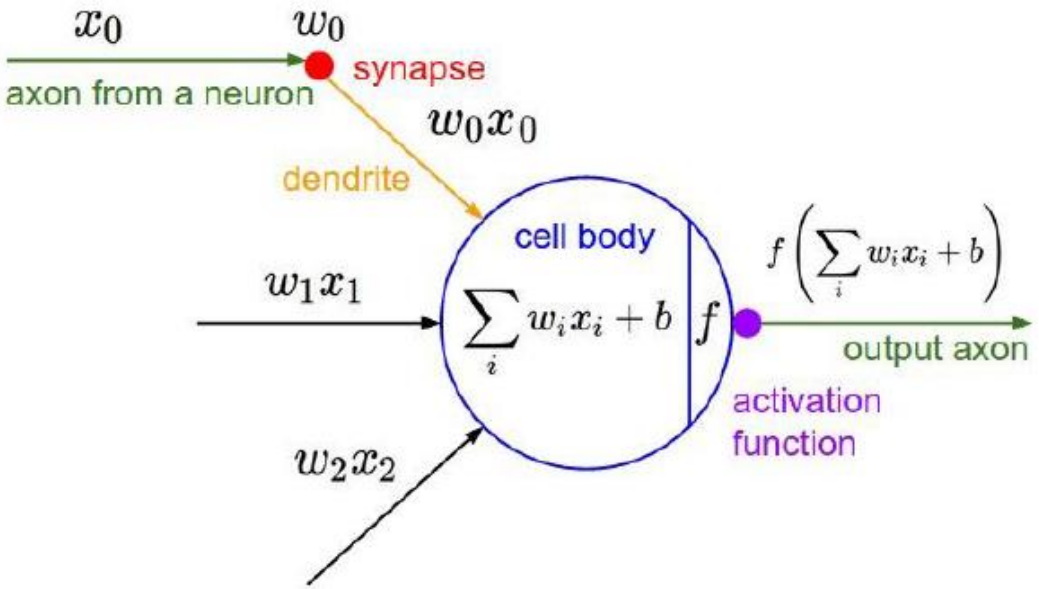


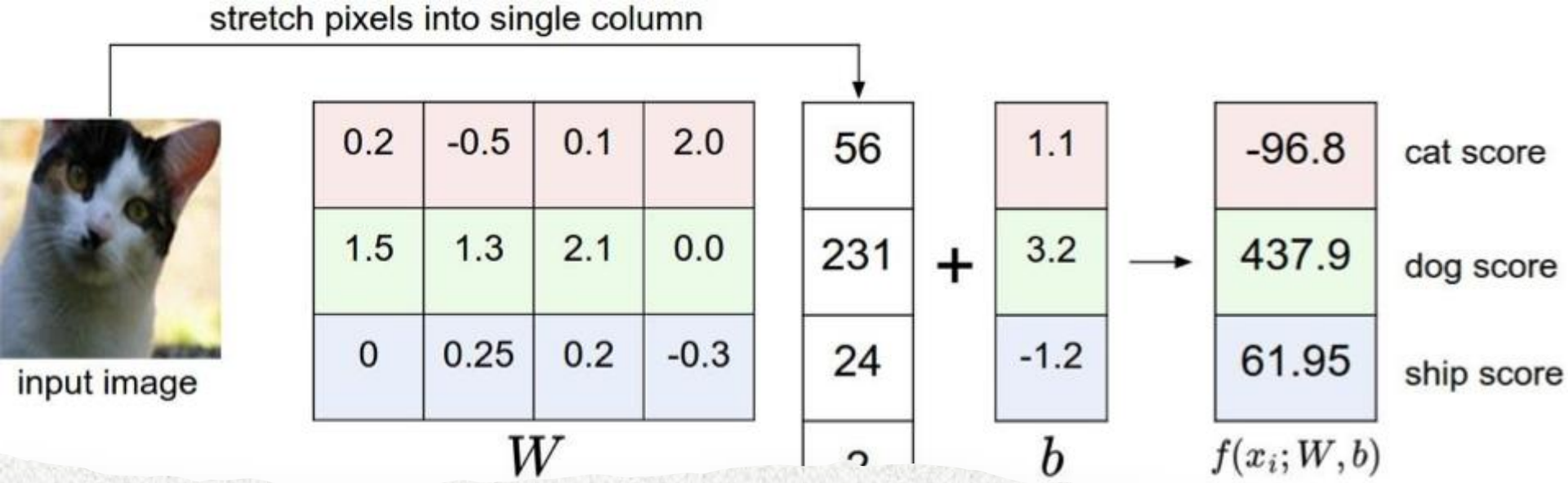


Recall Neural Network

- Neurons
  - accept information from multiple inputs,
  - transmit information to other neurons.
- Multiply inputs by weights along edges
- Apply some function to the set of inputs at each node

Recall Neural Network





We would love to learn

What is CNN ?

In deep learning, a convolutional neural network (CNN/ConvNet) is a class of deep neural networks most applied to analyze visual imagery. When we think of a neural network, we think about matrix multiplications, but that is not the case with ConvNet. It uses a special technique called Convolution. In mathematics, convolution is a mathematical operation on two functions that produces a third function that expresses how the shape of one is modified by the other. These networks are designed to work well for matrix-structured inputs (2D and 3D). The MLP network transforms the input data structure and transforms a 100x100 2D matrix into a 10000-dimensional vector. However, CNN does not change the input structure.

Applications of CNN

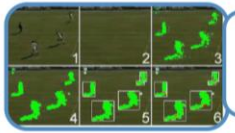
Computer Vision and related application

Natural Language Processing

Object Detection and Segmentation

Image Classification

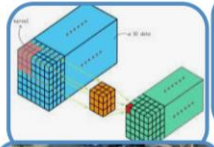
Speech Recognition



Video Processing

Low-resolution images

Limited Resource system

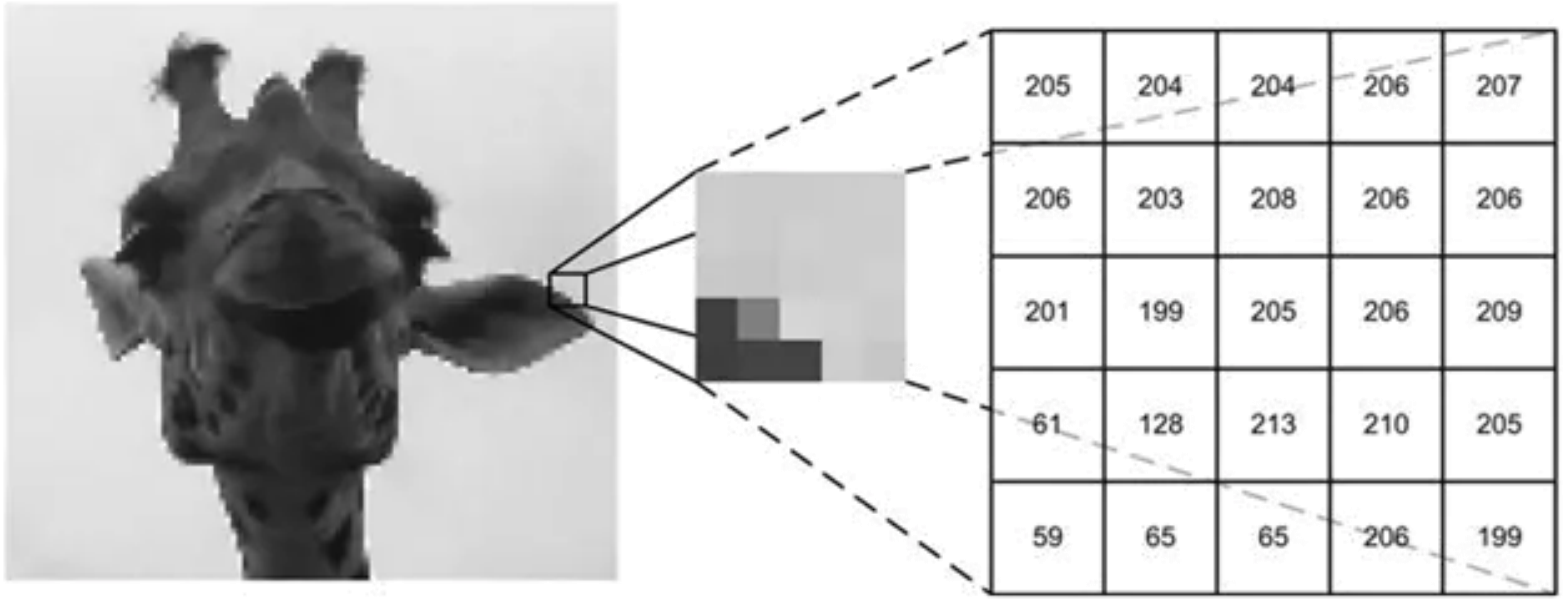


CNN for various dimensional data

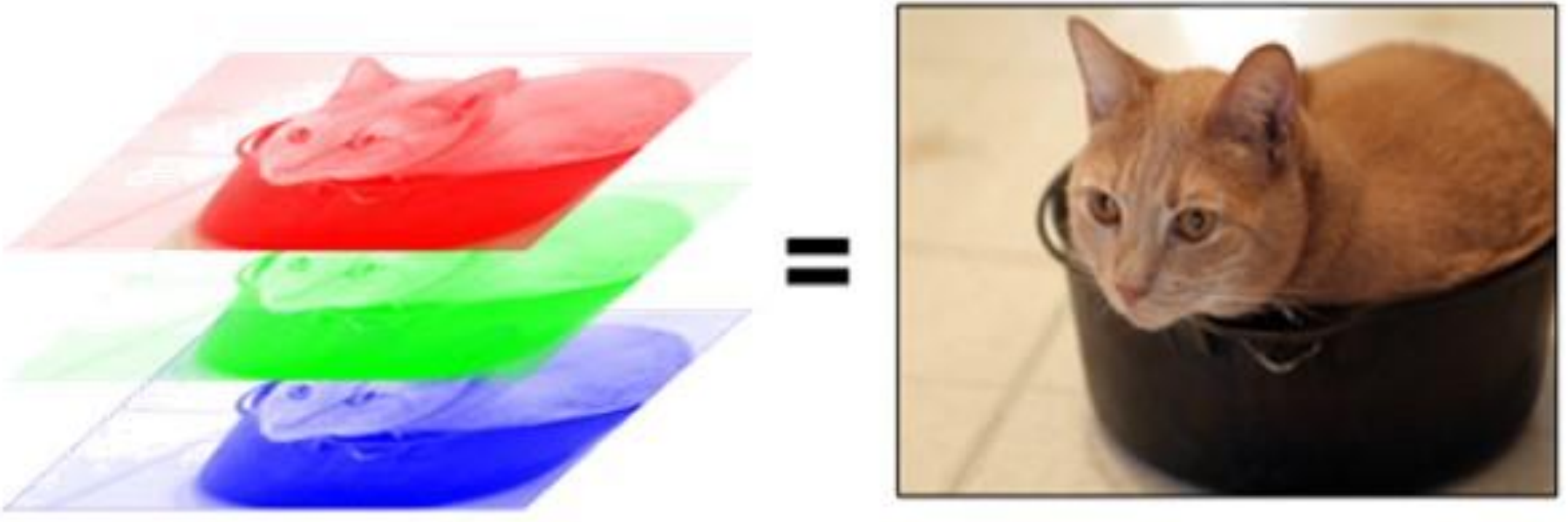
Object Counting

Convert image to array

In its simplest form, an image is a two-dimensional array or matrix of numbers, each equivalent to one pixel. A zoomed view of the image is also shown. You can see how the little squares fit together. **The values in the image or matrix are between 0 and 255. Zero is equivalent to absolute black, and 255 is equivalent to white**. The closer these values are to zero, the darker the pixel will be. The closer the pixel value is to 255, the brighter the pixel's color. Two types of grayscale and color images (RGB images) are usually given as input to the convolutional neural network. Color images have a slightly different structure. The color image consists of three pages. **These are red plate (R), green plate (G), and blue plate (B).** The specifications of each page are the same as the gray-level image, and the numerical values are between 0 and 255. A final color is obtained from the combination of three numbers. That is why color images are called RGB images. By combining these three pages, a color image is obtained.



0 = Black 128 = Mid–Gray 255 = White



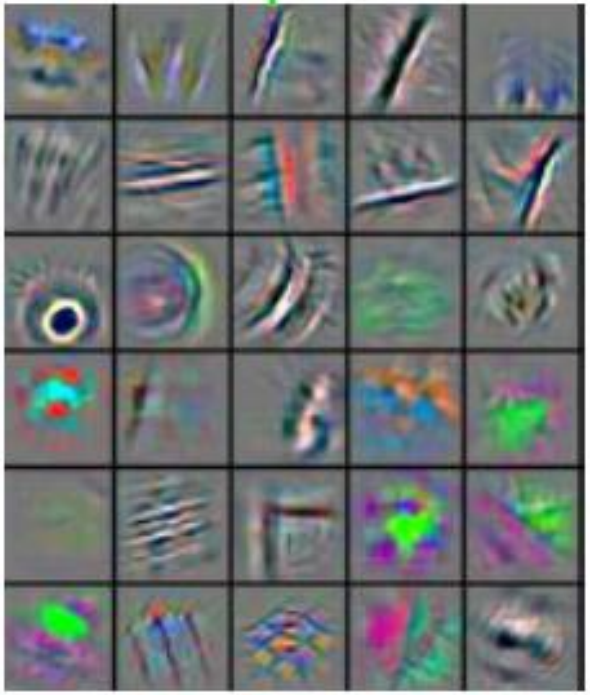
Low-Level Feature

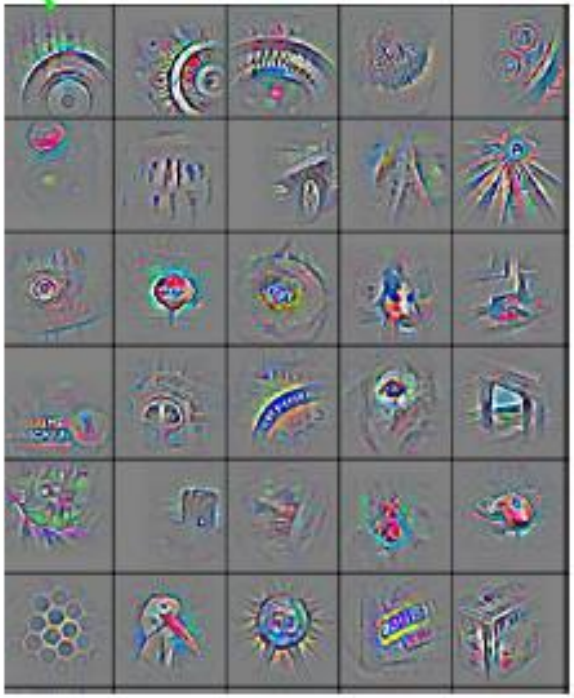
Mid-Level Feature

High-Level Feature

Trainable Classifier

"car"





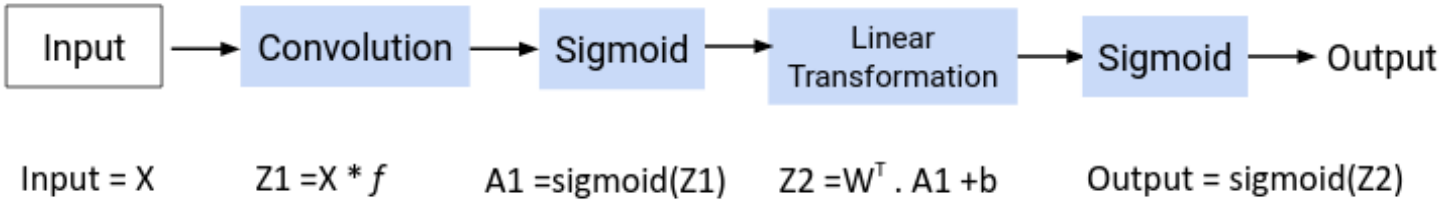
Tensors

Tensors are generalizations of matrices and can be represented as a multidimensional array. The number, vector, and matrix are zero, one-dimensional, and two-dimensional tensors, respectively. Note that tensors can be indexed just like matrices and vectors. Indexing allows us to access all elements of a tensor. A gray-level image is equivalent to a 2D tensor, **and a color image is equivalent to a 3D tensor.**

A bit (binary unit) is the fundamental unit of a computer, and each bit can take on one of two values—0 or 1. A single byte consists of 8 bits. In case you're wondering, the [0, 255] range comes from the pixel value being 8 stored in 8 bits, ($2^8 - 1 = 255$). However, we could also have a 16-bit data value. In coloured images, we can have either 8-bit, 16-bit, 24-bit, or 30-bit values, but we usually use 24-bit values since we have three coloured pixels, RGB, and each has an 8-bit data value.

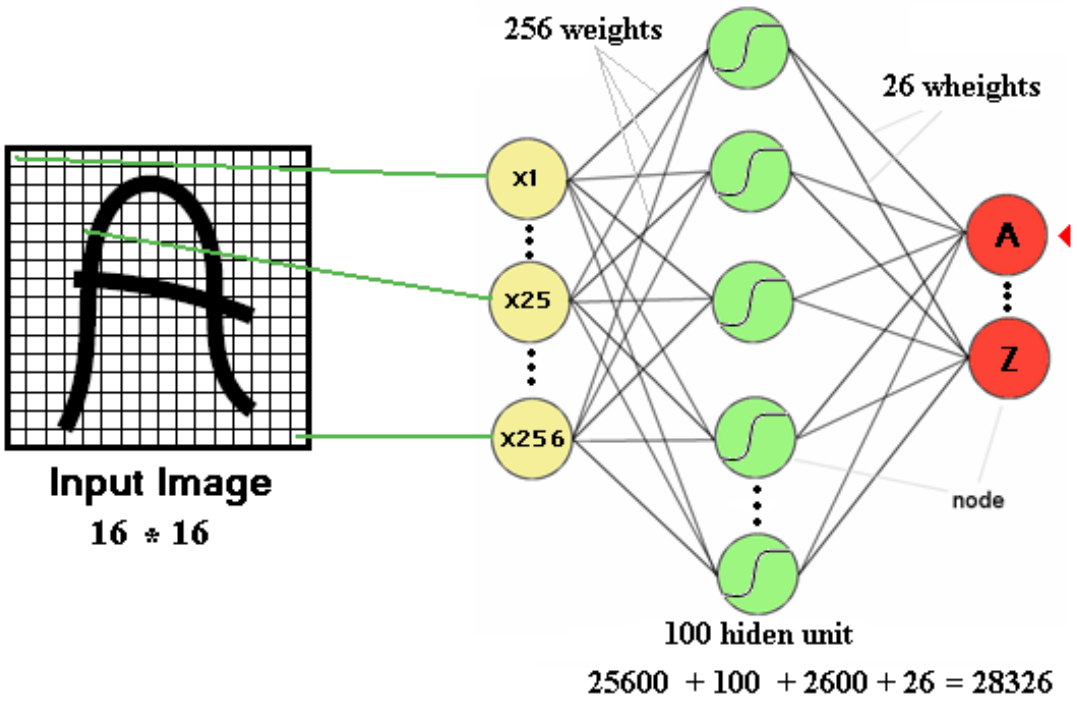
Suppose we have a grayscale image with a size of $512 \times 512 \times 1$ (height × width × channel). We can store it in a two-dimensional tensor (matrix), $\mathbb{R}^{h \times w \times 1}$, where each i and j value is a pixel with some intensity. To store this image on our disk, we need $512 \times 512 = 262144$ bytes.

The convolution operator takes a convolutional kernel or filter and slides it over the input image or matrix. The kernel or filter moves over the image or scans the input image. The filter first goes through each row column by column and then goes down one row and moves forward column by column again.



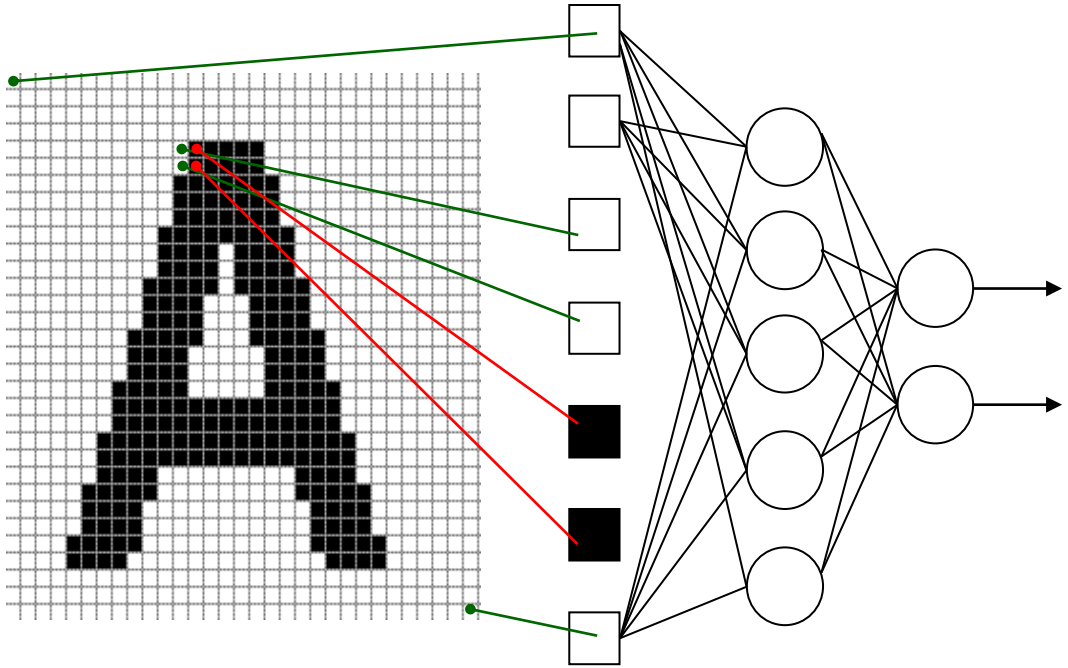
We have too many parameters

Why CNN



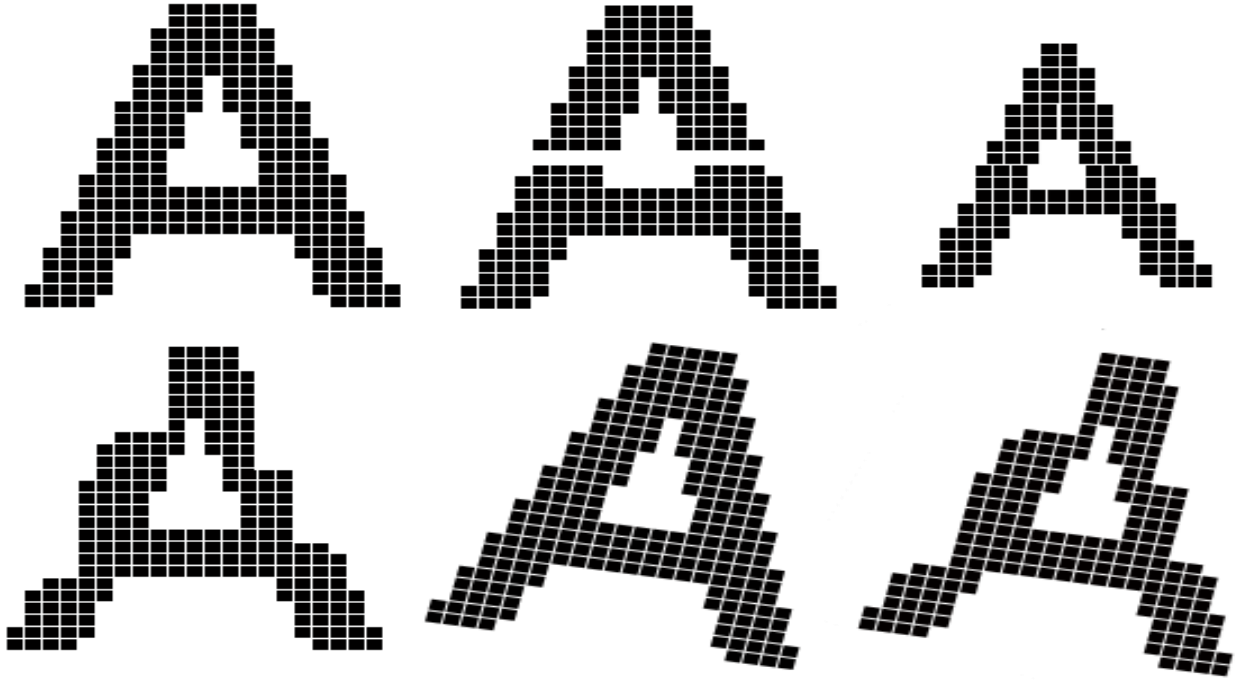
NN is not invariant wrt transition, rotation, resizing,

Why CNN



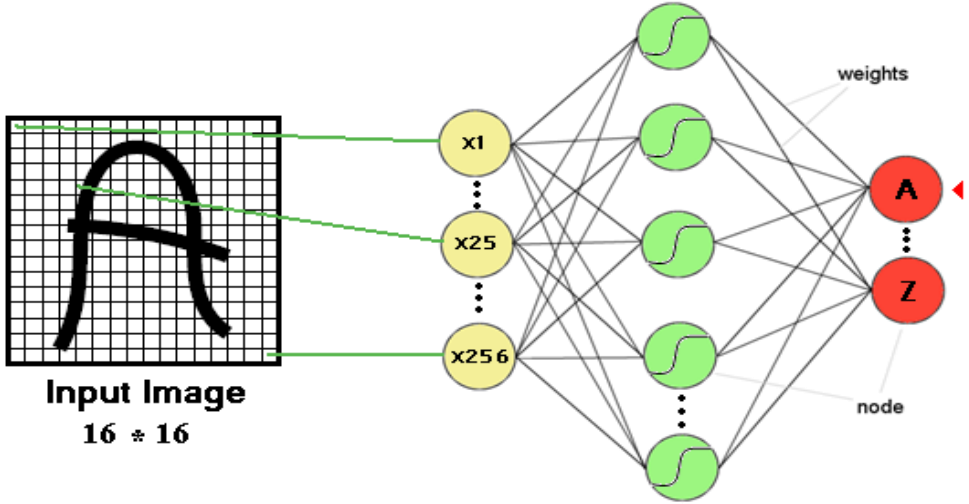
Resizing and rotation will change the weights

Why CNN



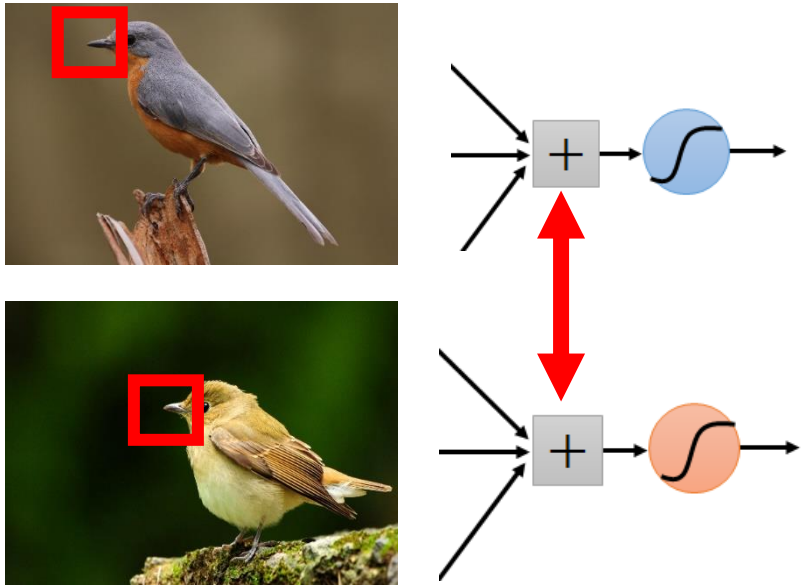
We want to consider 2-dimensional array for image analysis

Why CNN



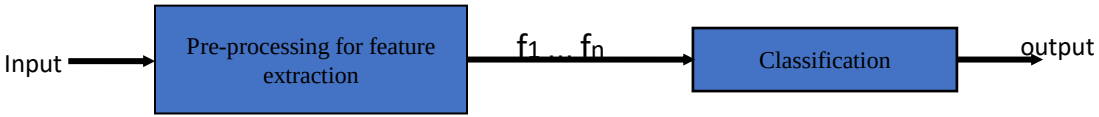
In the images of a specific class (for example, birds), important patterns (such as bird's tip) that play an important role in class recognition are located at different points of the sample images, which can be extracted with a small recognition unit.

Why CNN

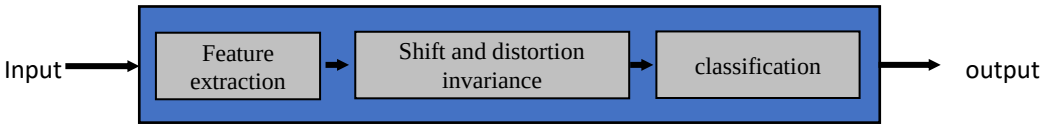


We can use the same weights to extract the bird’s tip feature.

ANN ☯


Input
Pre-processing for feature extraction
f1 … fn
Classification
output

CNN ☯


Input
Feature extraction
Shift and distortion invariance
classification
output

Convolution

What this means is that we have a function, f , which is our input and a function, g , which is our kernel. By convolving them, we receive an output (sometimes called a feature map).

In mathematics, we write convolutions as follows:

$$(f * g)(x) = \int f(t)g(t - x)\mathrm{d}t$$

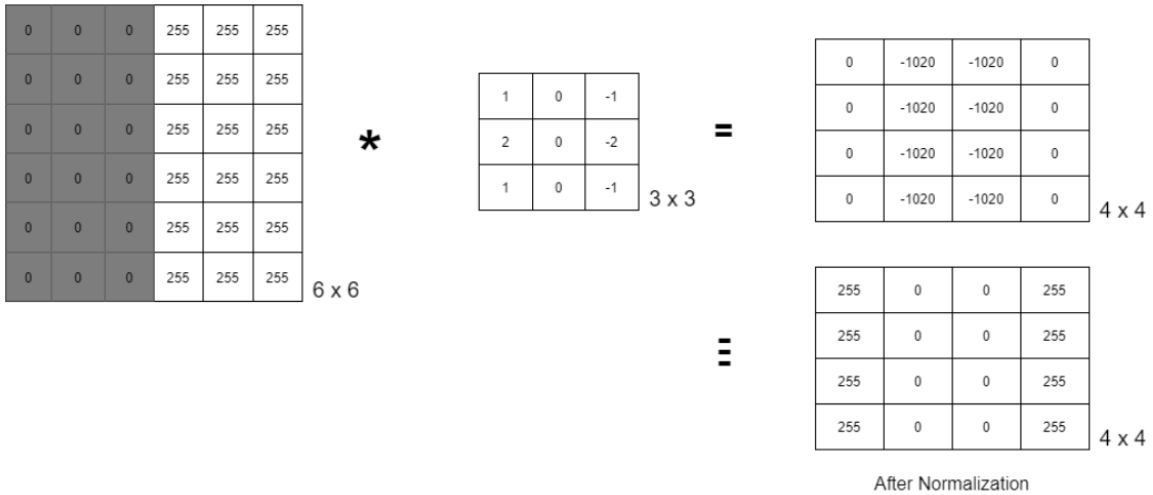
However, in CNNs, we usually use discrete convolutions, which are written as follows:

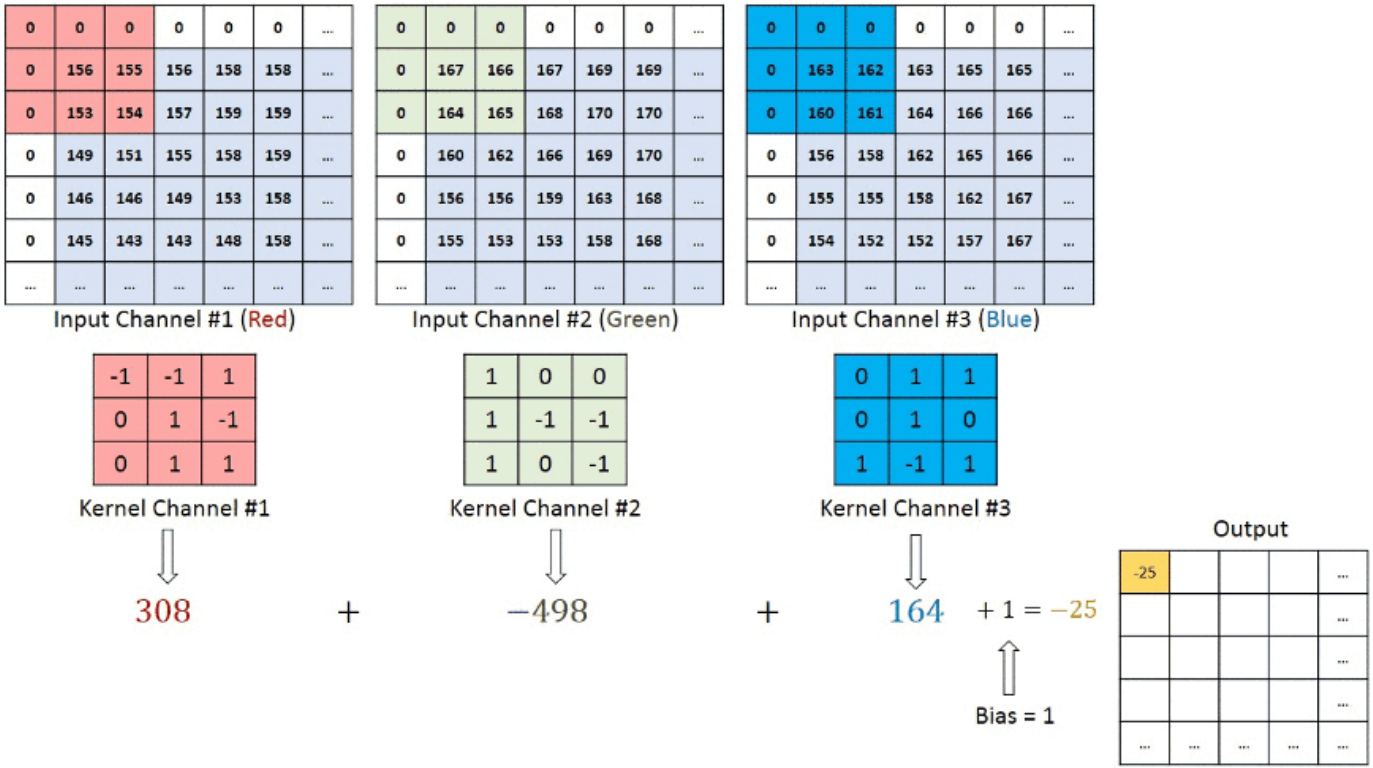
$$(f * g)(x) = \sum_t f(t)g(t - x)$$

As you might have guessed from the name itself, convolution convols/merges two functions or information to produce a third function/information. **In a convolutional neural network, the hidden layers include layers that perform convolutions.**

Let's consider this gray-scaled image of 6×6 dimensions. Since the machine can only understand binary language, this image would appear to be a 6×6 matrix with all 0s on the left half and all 255 on the right half, since it is a grayscale image. The RHS 3×3 matrix is known as kernel/filter/mask/ operator, and the $*$ is a convolution operator (here it is the Sobel edge detector doing component-wise multiplication and addition); we now have a 4 times 4 matrix. When we normalize this matrix, we receive an image with an edge highlighted.

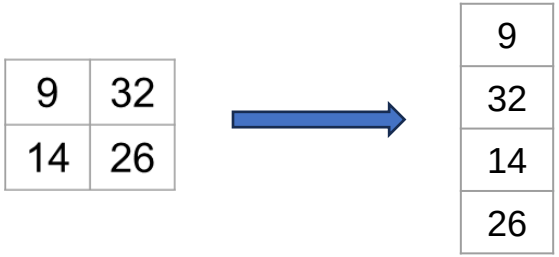
- Formula: (n x n) * (k x k) => (n – k + 1) x (n – k + 1)

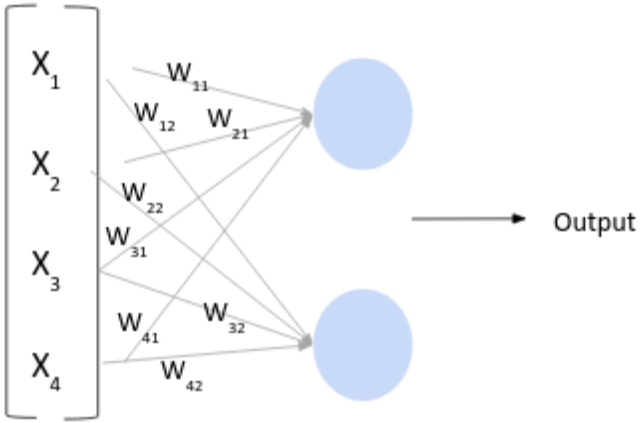



0 0 0 0 0 0 ...
0 156 155 156 158 158 ...
0 153 154 157 159 159 ...
0 149 151 155 158 159 ...
0 146 146 149 153 158 ...
0 145 143 143 148 158 ...
... ...
Input Channel #1 (Red)
0 0 0 0 0 0 ...
0 167 166 167 169 169 ...
0 164 165 168 170 170 ...
0 160 162 166 169 170 ...
0 156 156 159 163 168 ...
0 155 153 153 158 168 ...
... ...
Input Channel #2 (Green)
0 0 0 0 0 0 ...
0 163 162 163 165 165 ...
0 160 161 164 166 166 ...
0 156 158 162 165 166 ...
0 155 155 158 162 167 ...
0 154 152 152 157 167 ...
... ...
Input Channel #3 (Blue)
-1 -1 1
0 1 -1
0 1 1
Kernel Channel #1
1 0 0
1 -1 -1
1 0 -1
Kernel Channel #2
0 1 1
0 1 0
1 -1 1
Kernel Channel #3
308
+
−498
+
164
+ 1 = −25
Bias = 1
Output
-25 ... ...
... ...
... ...
... ...
... ...

Fully Connected Layer

The convolution layer has extracted some valuable features from the data. These features are sent to the fully connected layer that generates the final results. The fully connected layer in a CNN is nothing but the traditional neural network. The output from the convolution layer was a 2D matrix. Ideally, we want each row to represent a single input image. The fully connected layer can only work with 1D data. Hence, the values generated from the previous operation are converted into a 1D format. Once the data is converted into a 1D array, it is sent to the fully connected layer. These individual values are treated as separate features representing the image. The fully connected layer performs two operations on the incoming data – a linear and non-linear transformation.





$$X = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix}$$

Input Data

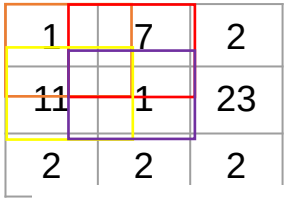
$$W = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \\ W_{31} & W_{32} \\ W_{41} & W_{42} \end{bmatrix}$$

Randomly Initialized Weight Matrix

$$b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Randomly Initialized bias Matrix

Example



1	1
0	1

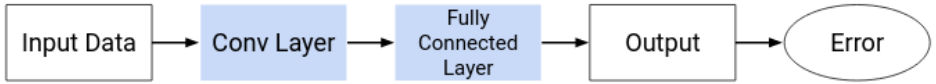
9	32
14	26

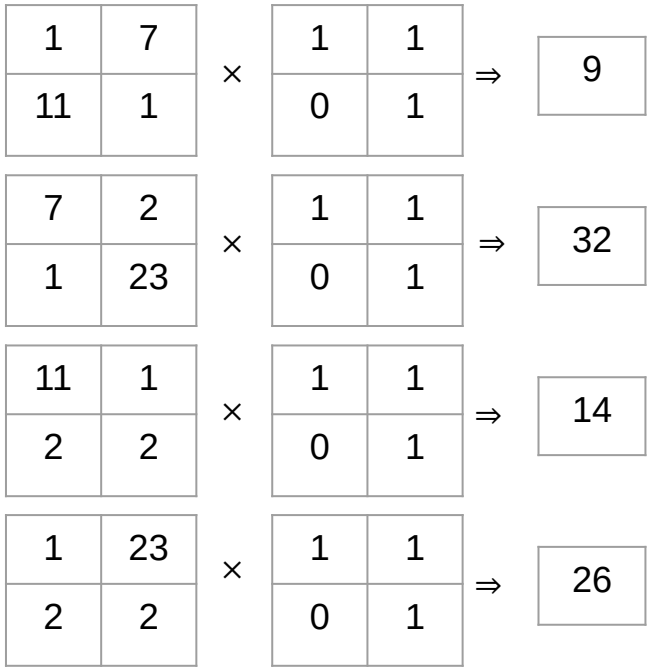
Dimensions

Input (Image) = 3×3

Filter = 2×2

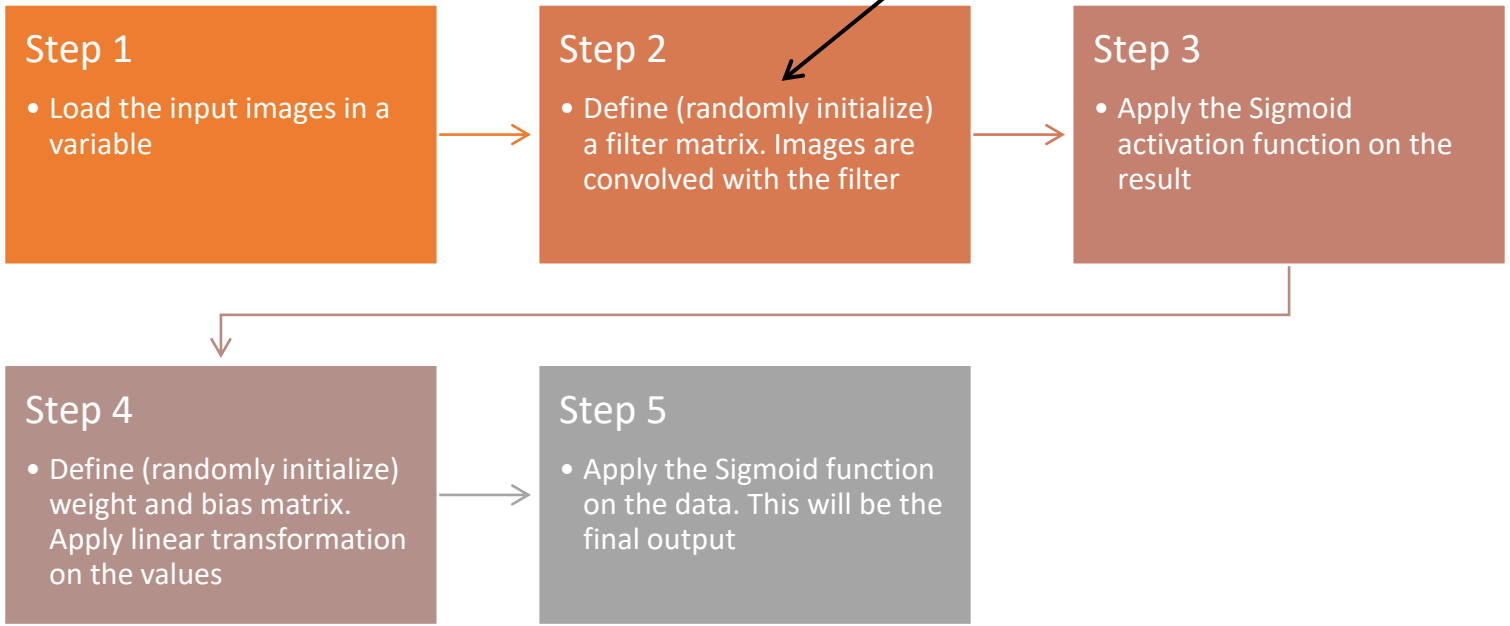
Output = $(3 - 2 + 1) \times (3 - 2 + 1)$

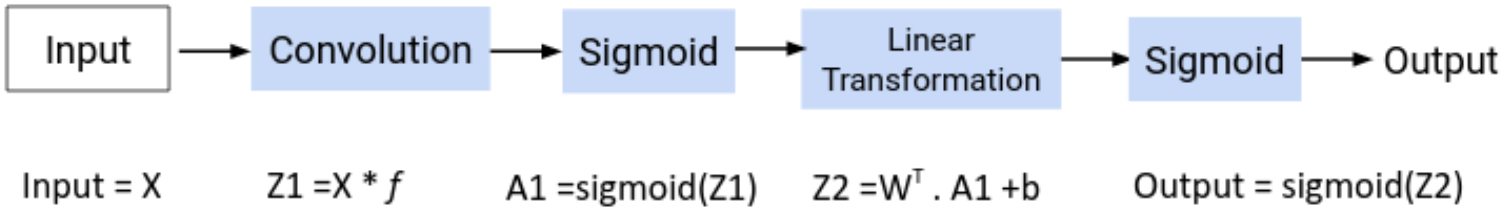



1 7
11 1
×
1 1
0 1
⇒
9
7 2
1 23
×
1 1
0 1
⇒
32
11 1
2 2
×
1 1
0 1
⇒
14
1 23
2 2
×
1 1
0 1
⇒
26

Forward Propagation

We can even learn



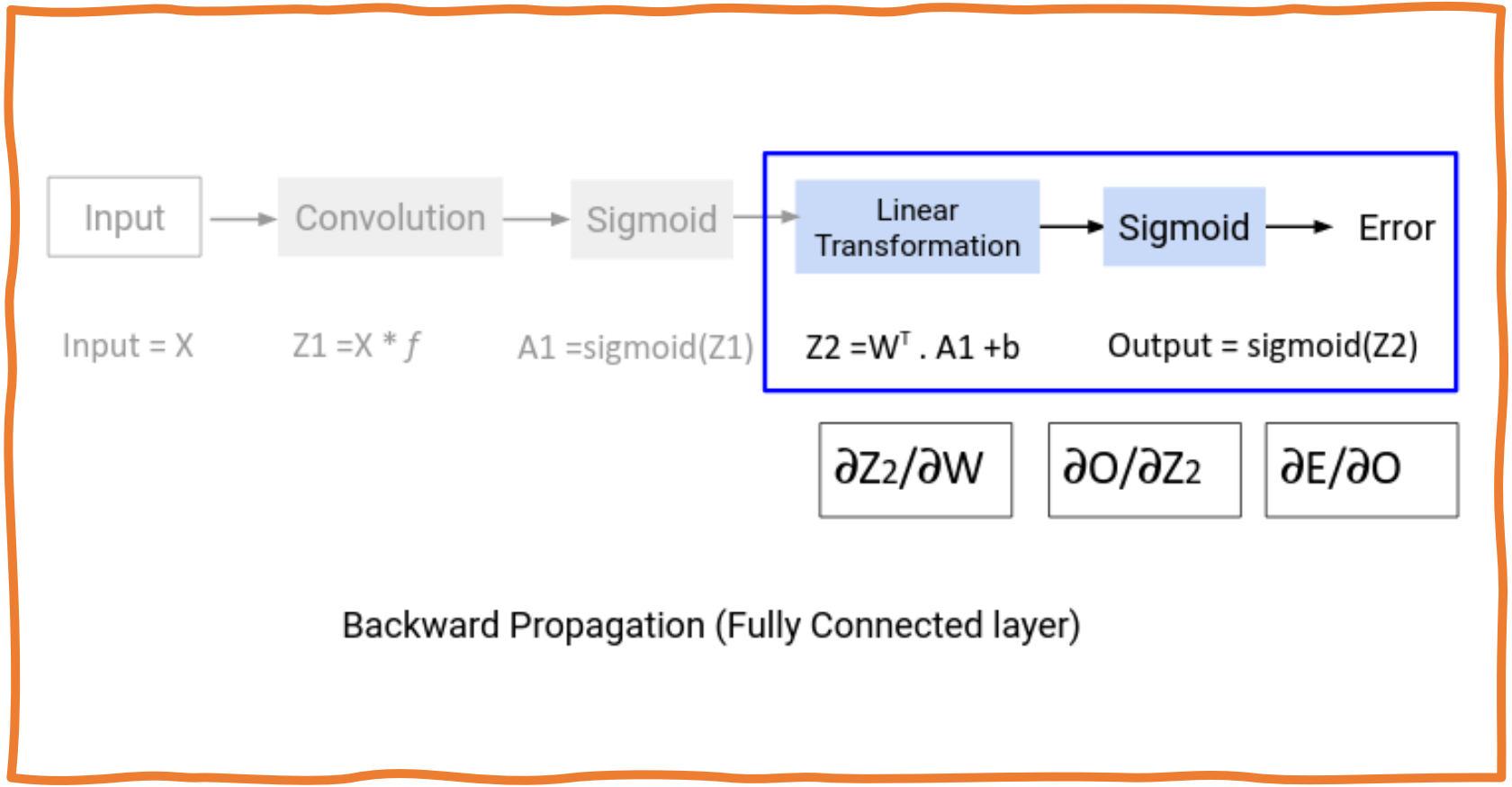

Input
Convolution
Sigmoid
Linear Transformation
Sigmoid
Output
Input = X
Z1 =X * f
A1 =sigmoid(Z1)
Z2 =W^T . A1 +b
Output = sigmoid(Z2)

Backward Propagation

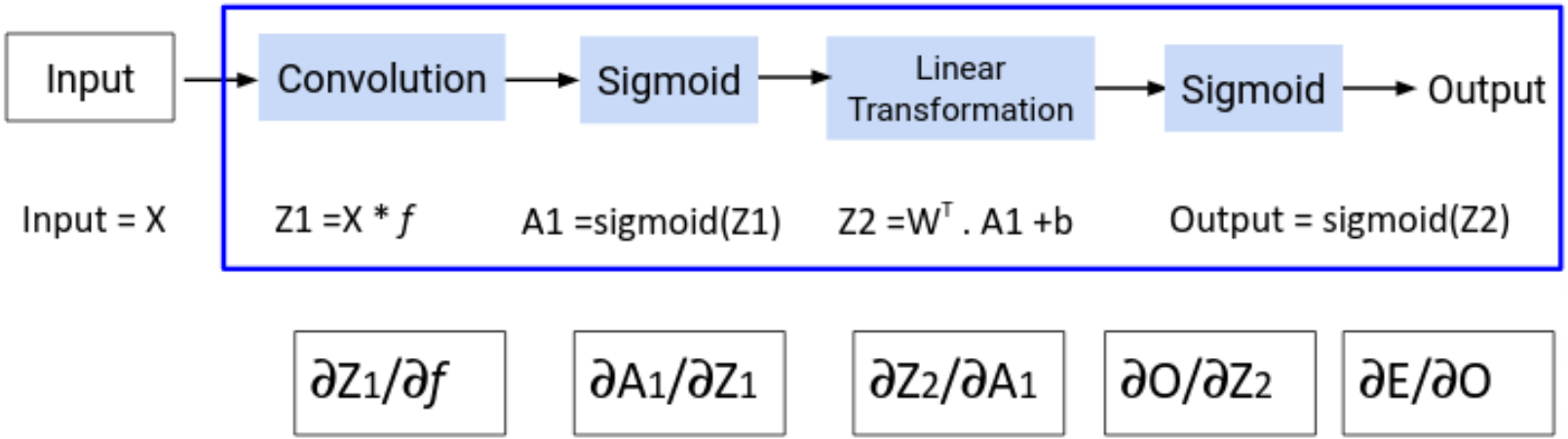
We randomly initialized the weights, biases, and filters during the forward propagation process. These values are treated as parameters from the CNN algorithm.

In the backward propagation process, the model tries to update the parameters such that the overall predictions are more accurate. Based on the value of the gradient, we can determine the updated parameter values.

$$new\_parameter = old\_parameter - (learning_{rate} \times gradient\_of\_parameter)$$



Backward Propagation (Fully Connected layer)



Backward Propagation (Convolution layer)

Pooling Layers

Pooling Layers, also known as downsample layers, are an essential component of CNNs used in deep learning. They are responsible for reducing the spatial dimensions of the input data in terms of width and height while retaining the most essential information. Pooling Layers divide the input data into small regions, called **pooling windows** or **receptive fields**, and perform an aggregation operation, such as taking the maximum or average value within each window.
This aggregation reduces the size of the feature maps, resulting in a compressed representation of the input data.

This works somewhat like the convolution operation, except it reduces the size of the feature map by sliding a window across it and either averages all the values inside each window at each step or outputs the maximum value. The pooling operation differs from convolution because it has no parameters, so it cannot be learned or tuned.

1	8	4	3
6	5	9	2
2	5	6	5
8	9	4	3

Max pooling

8	9
9	6

Sum pooling

20	18
24	18

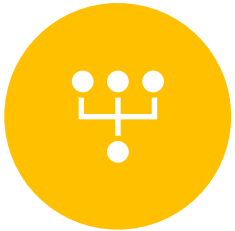
Average pooling

5	4.5
6	4.5

Why Pooling Layers is Important

Dimensionality Reduction: By reducing the spatial dimensions of the input data, Pooling Layers help in reducing the number of parameters and computational complexity of the subsequent layers in the model.

Translation Invariance: Pooling Layers provide a form of translation invariance by extracting the most important features from different spatial locations, making the model more robust to variations in the position of the features.



Feature Hierarchy: Pooling Layers help create a hierarchical representation of the input data, where lower-level features are combined to form higher-level features, capturing abstract representations.

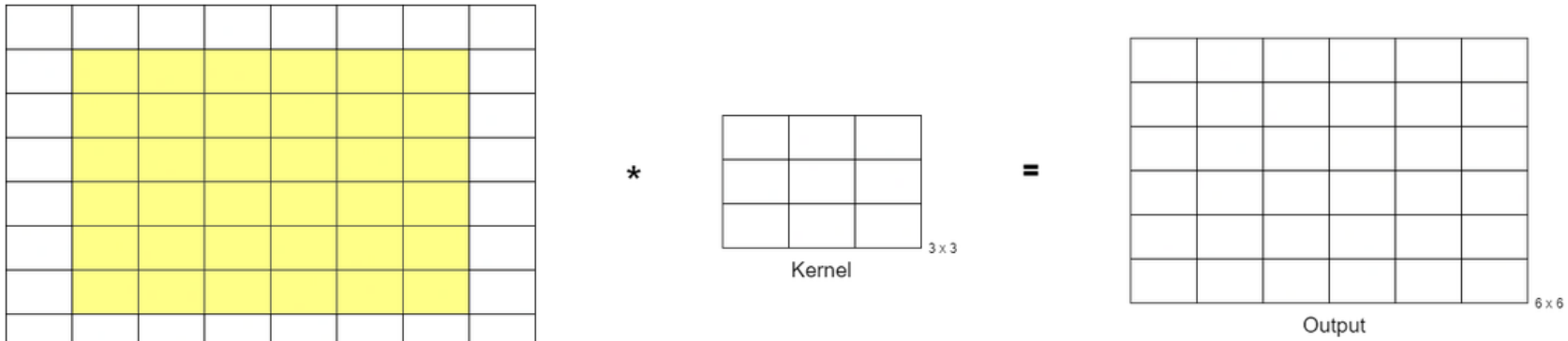
Regularization: Pooling Layers can act as a form of regularization by reducing overfitting. By aggregating information from local regions, the model focuses on the most relevant information and ignores minor variations.

What is Padding?

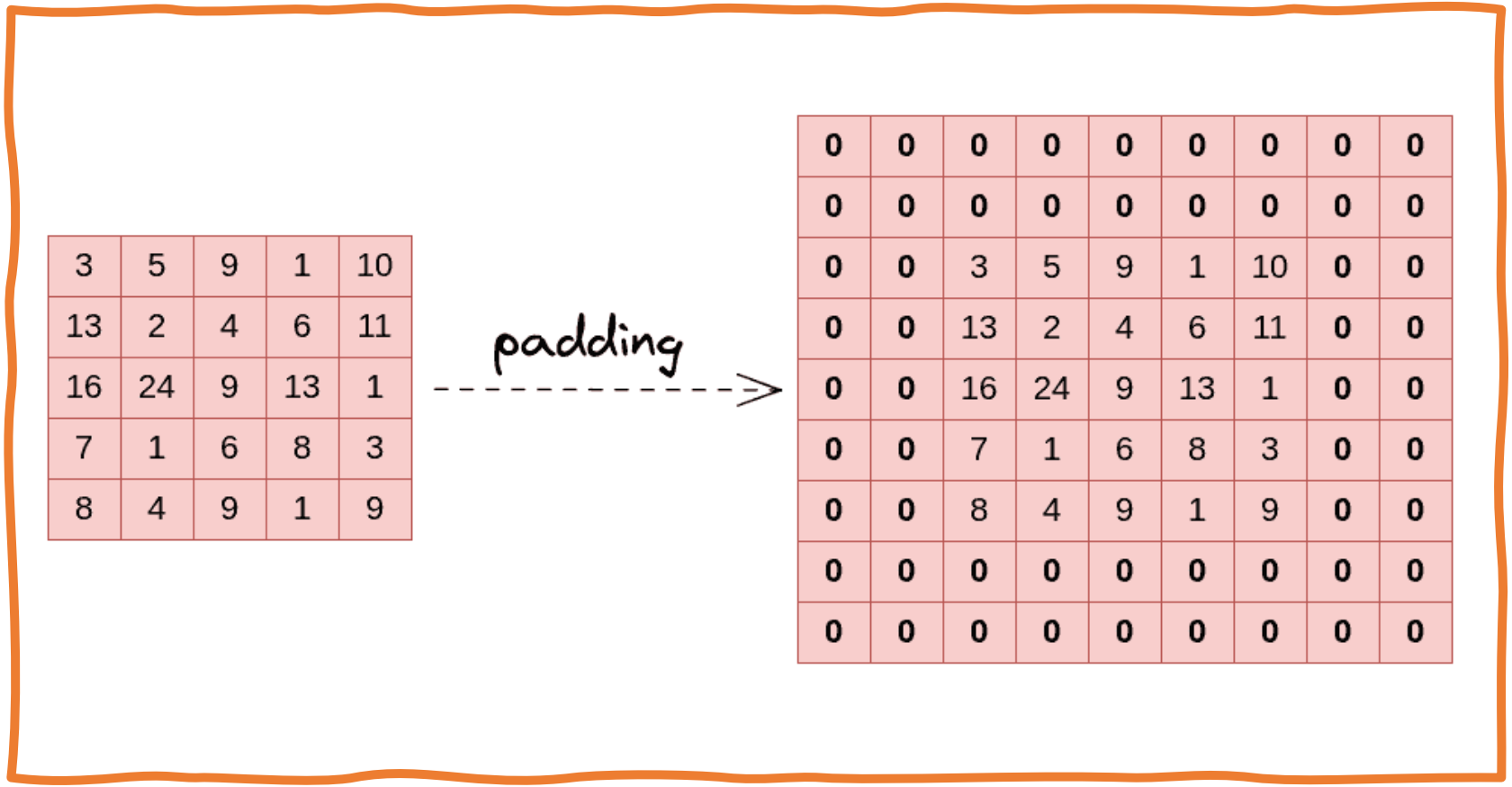
In CNNs, padding refers to the process of adding extra pixels to the input image before applying convolution. Acting as a protective boundary around the image, these additional pixels allow the network to retain more spatial information. This is the main use of padding in CNN. you apply padding symmetrically on all sides of the input image. The most common strategy is zero padding in CNN, where you add a bunch of zeros around your input feature map .

What if we want the output matrix to have the same dimensions as the input matrix?

What if we want the output matrix to have the same dimensions as the input matrix i.e. $(\boldsymbol{n} \times \boldsymbol{n})$. If you substitute $\boldsymbol{n} = \boldsymbol{6}$ from the above formula, you will get an output matrix of size $\boldsymbol{4} \times \boldsymbol{4}$. If you want the output matrix to be of size $\boldsymbol{6} \times \boldsymbol{6}$ then it is quite obvious that the input matrix should be of size $\boldsymbol{8} \times \boldsymbol{8}$, so to change the dimensions of the input matrix, there is a concept of padding. If a pad one extra layer to each side then the initial dimension would be increased by 2 i.e. now $\boldsymbol{n} = \boldsymbol{8}$ which will give the output matrix dimension to be 6.



Padding, p = 1


3 5 9 1 10
13 2 4 6 11
16 24 9 13 1
7 1 6 8 3
8 4 9 1 9
padding
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 3 5 9 1 10 0 0
0 0 13 2 4 6 11 0 0
0 0 16 24 9 13 1 0 0
0 0 7 1 6 8 3 0 0
0 0 8 4 9 1 9 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0

Valid Padding / No Padding

- Valid padding is the absence of extra pixels added to the borders of the input feature map. Thus, Valid Padding is essentially the same as no padding.
- In this strategy, the convolution is performed only on the pixels which completely overlap with the filter. As a result, its output feature maps are smaller than the input. It gives rise to a more pronounced downsampling effect. Valid Padding is often used when reducing spatial dimensions and extracting high-level features are desirable. Object classification tasks, or those requiring reduced computational complexity benefit from this technique. While Valid Padding leads to a loss of spatial information, it can be advantageous when capturing global context and reducing the computational burden.
- Understanding the applications of Same Padding, Causal Padding, and Valid Padding empowers CNN practitioners to make informed decisions. It's crucial when designing network architectures for specific tasks. The appropriate padding technique allows you to strike a balance between spatial preservation and efficiency.

Same Padding / Zero Padding

This popular technique ensures size parity between the output feature maps and the input image. It achieves this by padding the input borders in a way that allows the convolutional filter to stride evenly across the image.

Same Padding allows every pixel in the input to be at the center of the filter at least once. Its application preserves spatial information and avoids border effects. Here, padding size depends directly on the size of the filter.

Same Padding is particularly valuable when spatial preservation is crucial. Tasks requiring precise object localization, or those which need to maintain the resolution of feature maps benefit from it. By maintaining spatial dimensions throughout the network, Same Padding ensures that positional details and fine-grained information are accurately captured.

Constant Padding / Causal padding

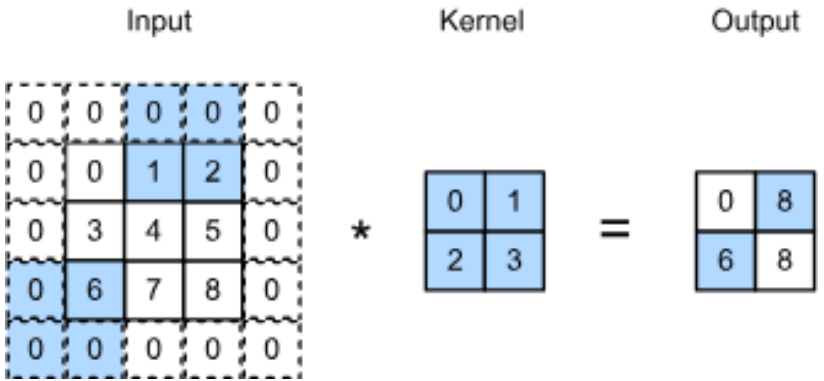
Causal padding is useful in sequence-related tasks. Popular examples include natural language processing (NLP) and time series analysis. Here, the padding is applied asymmetrically: you add extra values only to the left side of the sequence. This is considered the “past” of the sequence. Thus, this technique maintains the causality of the data. It ensures that each output element only depends on past or current input elements.

Causal padding is essential when dealing with tasks that require predicting future values based on past or present observations. It prevents the network from accessing future information during training, making it suitable for tasks like language modeling or forecasting, where temporal dependencies must be preserved.

Check out for Reflection Padding and Replication Padding, also

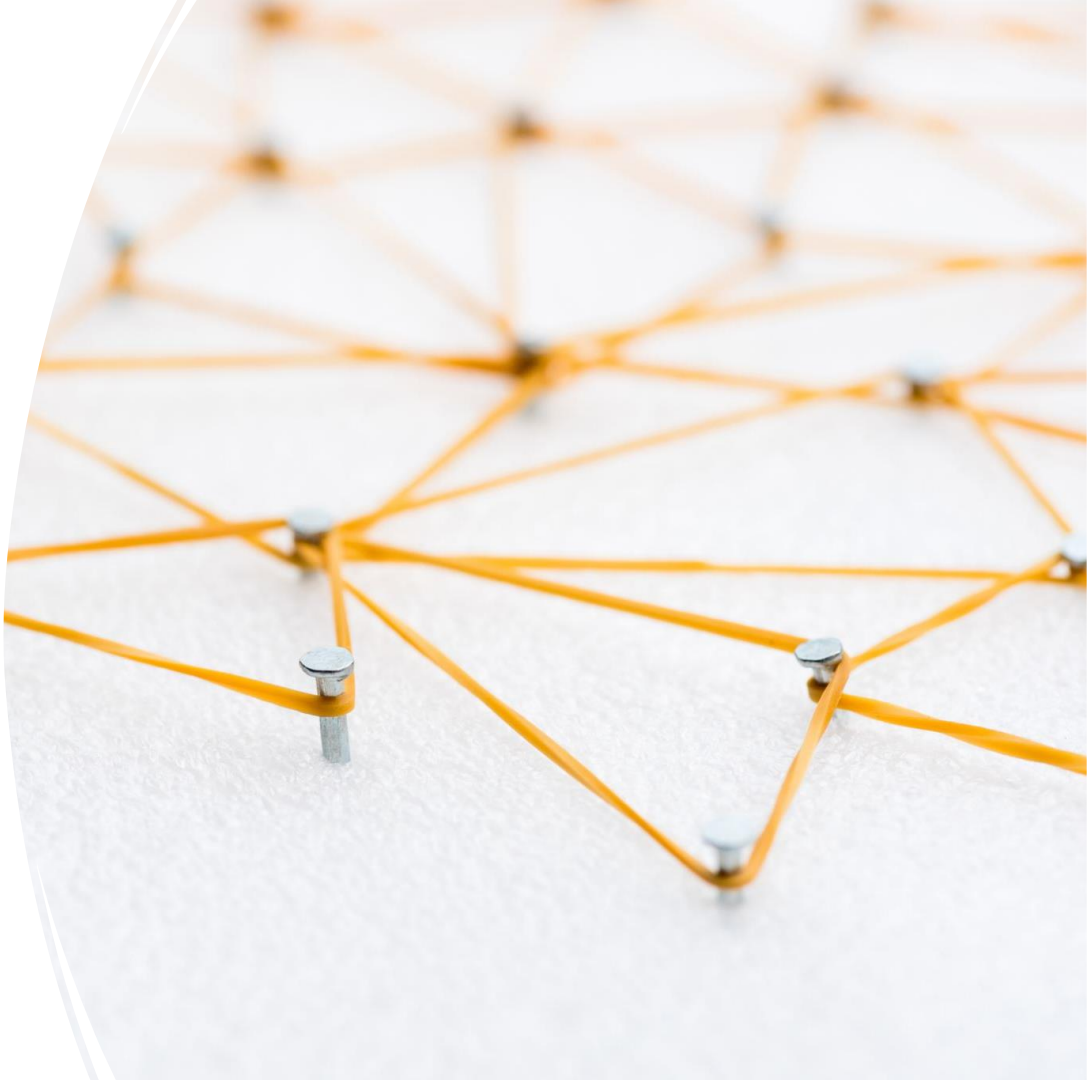
Strided convolution

When computing the cross-correlation, we start with the convolution window at the upper-left corner of the input tensor and then slide it over all locations both down and to the right. We defaulted to sliding one element at a time. However, sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one element at a time, skipping the intermediate locations. This is particularly useful if the convolution kernel is large since it captures a large area of the underlying image.



Feature Map

The main core of the CNN network is the convolution layer, which accounts for most convolutional neural network calculations. Each convolution layer in the convolution neural network contains a set of filters, and the output is made from the convolution between the filters and the input layer. The output of the convolutional layer is called Feature Map.



1. Usually, in the CNN, the convolutional layer is placed first. That is, before pooling, nonlinear scaling, etc. First, a convolutional layer is applied to the image with a low number of filters. You can see layer 1 with 32 filters in the figure below. Padding and stride are adjusted so that the size of the output picture does not decrease. The size of the output feature map is equal to 32x32x32.

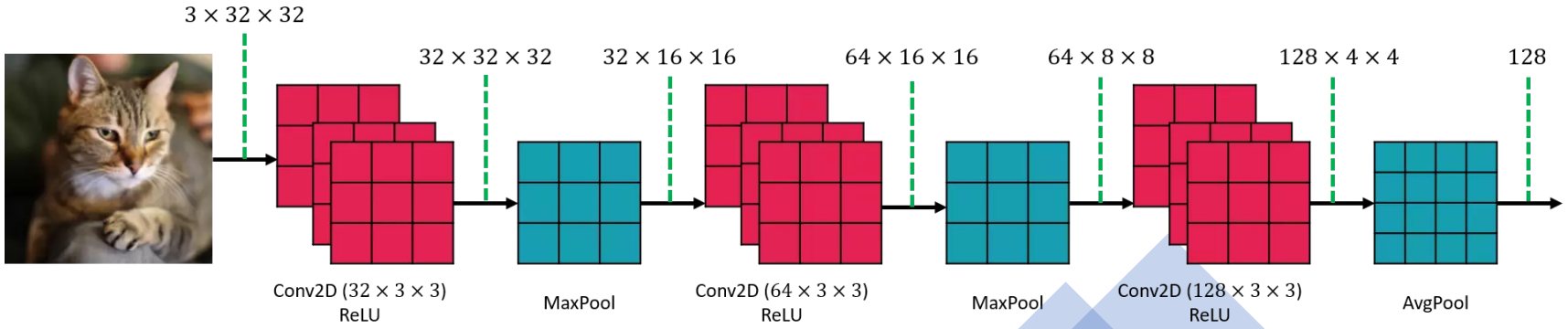
2. An excitation function is placed after the convolutional layer. It is normal, because if you remember, the stimulation function was applied to the neuron after weighting addition and addition with bias. As shown in the figure below, a ReLU function is applied to the feature map.

3. Now I want to add a pooling layer after the trigger function. This poling is a type of max poling. But I considered stride 2 to halve the size of the output picture. So the size of the output feature map will be 16x16x32.

4. So far I have applied a round of convolution, nonlinearity and pooling to the input image. Again, I apply all three of these. I considered the number of filters of the second convolutional layer to be 64. Max Pooling output feature map : 8x8x64

5. In the third step, a convolutional layer with 128 filters is considered and the stride is 2. After the convolutional layer, I put an excitation function. The output feature map of the stimulation function is equal to 4x4x128.

6. At the end, I have added an average pooling layer with a size of 4x4. The output feature of this layer will be equal to 1x128. That is, the input image was transformed into a vector of length 128 after several steps. This vector is the feature map vector of the input image.



Introduction to Keras

Keras is a deep-learning framework that provides a convenient way to define and train almost any kind of deep-learning model.

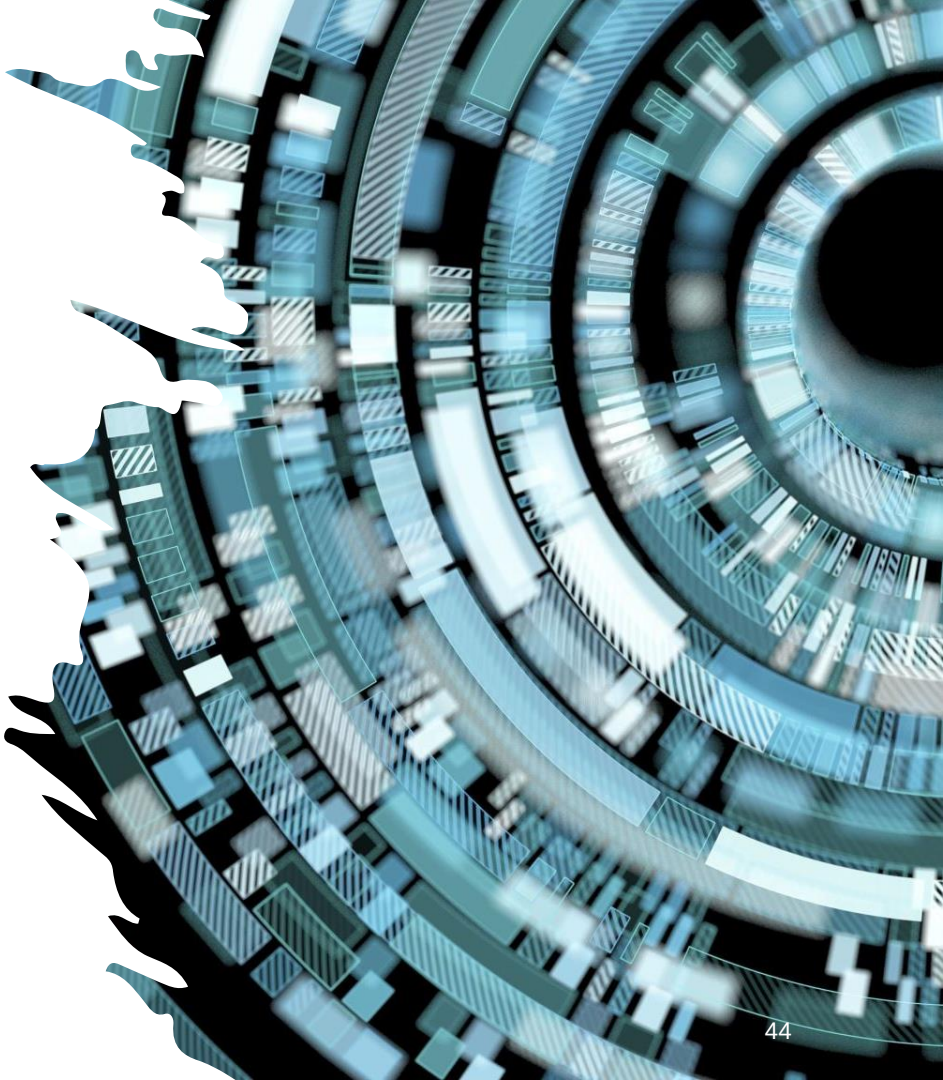
Keras has the following key features:

It allows the same code to run seamlessly on CPU or GPU.

It has a user-friendly (application programming interface) API that makes it easy v.

It has built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination.

It supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, etc. Keras is appropriate for building any deep-learning model, from a generative adversarial network to a neural Turing machine.



Developing a Deep NN with Keras

Step 1 - Define your training data: input tensors and target tensors.

Step 2 - Define a network of layers (or model) that maps your inputs to your targets.

Step 3 - Configure the learning process by choosing

- a loss function,
- an optimizer,
- and some metrics to monitor.

Step 4 - Iterate on your training data by calling the fit() method of your model.

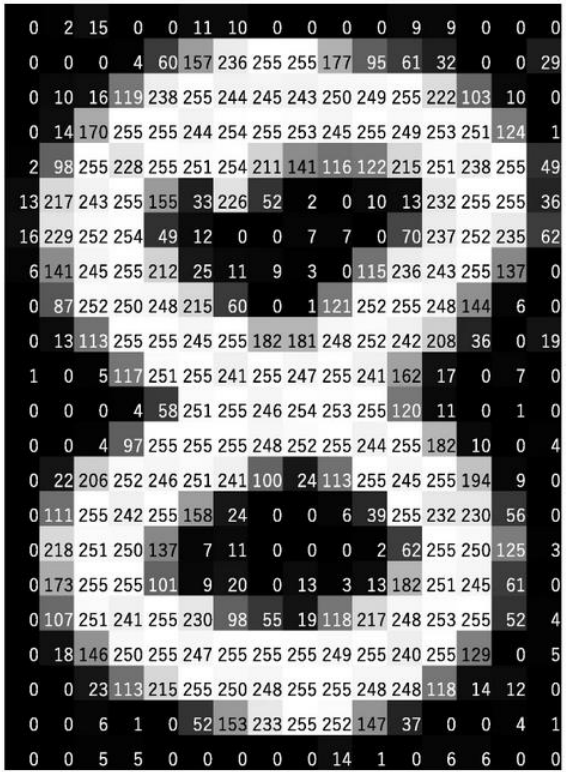
MNIST dataset

Images are 28 x 28 pixels

Represent input image as a vector $\mathbf{x} \in \mathbb{R}^{784}$
Learn a classifier $f(\mathbf{x})$ such that,

$$f : \mathbf{x} \rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

The problem we are trying to solve here is to classify grayscale images of handwritten digits (28 pixels by 28 pixels) into their 10 categories (0 to 9). The MNIST dataset is a classic dataset in the machine-learning community, which has been around almost as long as the field itself and has been intensively studied. It's a set of 60,000 training images, plus 10000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s. You can think of "solving" MNIST as the "Hello World" of deep learning---it is what you do to verify that your algorithms are working as expected. You can see some MNIST samples in the figure. The MNIST dataset comes preloaded in Keras, in the form of train and test lists, each of which includes a set of images (x) and associated labels (y). For instance, for this figure, the labels are 0, 2, 4, and 3.

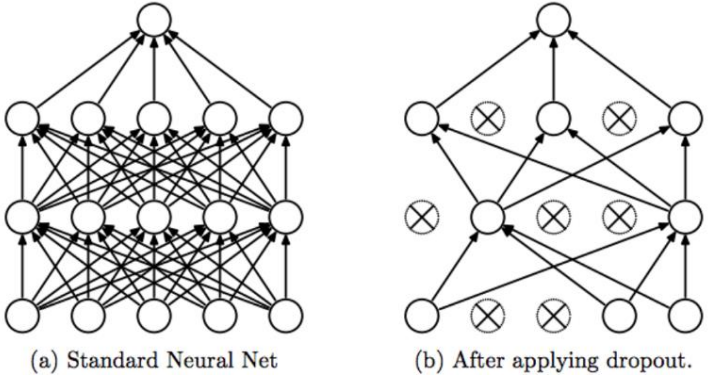


0	2	15	0	0	11	10	0	0	0	0	9	9	0	0	0
0	0	0	4	60	157	236	255	255	177	95	61	32	0	0	29
0	10	16	119	238	255	244	245	243	250	249	255	222	103	10	0
0	14	170	255	255	244	254	255	253	245	255	249	253	251	124	1
2	98	255	228	255	251	254	211	141	116	122	215	251	238	255	49
13	217	243	255	155	33	226	52	2	0	10	13	232	255	255	36
16	229	252	254	49	12	0	0	7	7	0	70	237	252	235	62
6	141	245	255	212	25	11	9	3	0	115	236	243	255	137	0
0	87	252	250	248	215	60	0	1	121	252	255	248	144	6	0
0	13	113	255	255	245	255	182	181	248	252	242	208	36	0	19
1	0	5	117	251	255	241	255	247	255	241	162	17	0	7	0
0	0	0	4	58	251	255	246	254	253	255	120	11	0	1	0
0	0	4	97	255	255	255	248	252	255	244	255	182	10	0	4
0	22	206	252	246	251	241	100	24	113	255	245	255	194	9	0
0	111	255	242	255	158	24	0	0	6	39	255	232	230	56	0
0	218	251	250	137	7	11	0	0	0	2	62	255	250	125	3
0	173	255	255	101	9	20	0	13	3	13	182	251	245	61	0
0	107	251	241	255	230	98	55	19	118	217	248	253	255	52	4
0	18	146	250	255	247	255	255	255	249	255	240	255	129	0	5
0	0	23	113	215	255	250	248	255	255	248	248	118	14	12	0
0	0	6	1	0	52	153	233	255	252	147	37	0	0	4	1
0	0	5	5	0	0	0	0	0	14	1	0	6	6	0	0

0	2	15	0	0	11	10	0	0	0	0	9	9	0	0	0
0	0	0	4	60	157	236	255	255	177	95	61	32	0	0	29
0	10	16	119	238	255	244	245	243	250	249	255	222	103	10	0
0	14	170	255	255	244	254	255	253	245	255	249	253	251	124	1
2	98	255	228	255	251	254	211	141	116	122	215	251	238	255	49
13	217	243	255	155	33	226	52	2	0	10	13	232	255	255	36
16	229	252	254	49	12	0	0	7	7	0	70	237	252	235	62
6	141	245	255	212	25	11	9	3	0	115	236	243	255	137	0
0	87	252	250	248	215	60	0	1	121	252	255	248	144	6	0
0	13	113	255	255	245	255	182	181	248	252	242	208	36	0	19
1	0	5	117	251	255	241	255	247	255	241	162	17	0	7	0
0	0	0	4	58	251	255	246	254	253	255	120	11	0	1	0
0	0	4	97	255	255	255	248	252	255	244	255	182	10	0	4
0	22	206	252	246	251	241	100	24	113	255	245	255	194	9	0
0	111	255	242	255	158	24	0	0	6	39	255	232	230	56	0
0	218	251	250	137	7	11	0	0	0	2	62	255	250	125	3
0	173	255	255	101	9	20	0	13	3	13	182	251	245	61	0
0	107	251	241	255	230	98	55	19	118	217	248	253	255	52	4
0	18	146	250	255	247	255	255	255	249	255	240	255	129	0	5
0	0	23	113	215	255	250	248	255	255	248	248	118	14	12	0
0	0	6	1	0	52	153	233	255	252	147	37	0	0	4	1
0	0	5	5	0	0	0	0	0	14	1	0	6	6	0	0

A computer sees an image as an array of numbers. The matrix on the right contains numbers between 0 and 255, each of which corresponds to the pixel brightness in the left image. Both are overlaid in the middle image. The source image was downloaded via http://yann.lecun.com/exdb/mnist

DNN with dropout-See MNIST



(a) Standard Neural Net (b) After applying dropout.

https://www.appsilon.com/post/r-keras-mnist

- A common concern in this type of network is overfitting (error on test data deviates considerably from training error). We want our model to achieve high generalization (low test error). There are several ways to regularize, such as introducing weight penalties (e.g., L1, L2), early stopping, and weight decay.
- The dropout method is one simple and efficient way to regularize our model. Dropout means that nodes and their connections will be randomly dropped with **probability p** during training. This way, an ensemble of thinned sub-networks will be trained and averaged for predictions

Let's have some examples with R and Python

classical CNN networks

- LeNet-5-1998
- AlexNet-2012
- VGG-2014

Residual Networks (ResNets)-2015

Inception network (GoogleNet)-2015

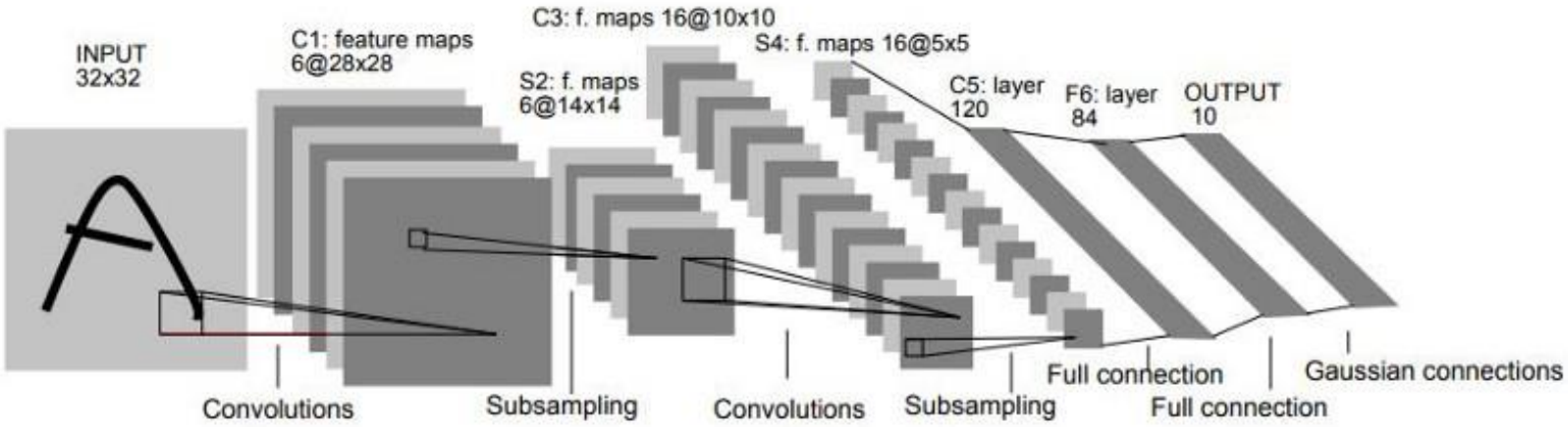
Bounding box prediction (YOLO)-2015

YOLO: You Only Look Once

In general, what the architecture learns can be demonstrated with the following flow

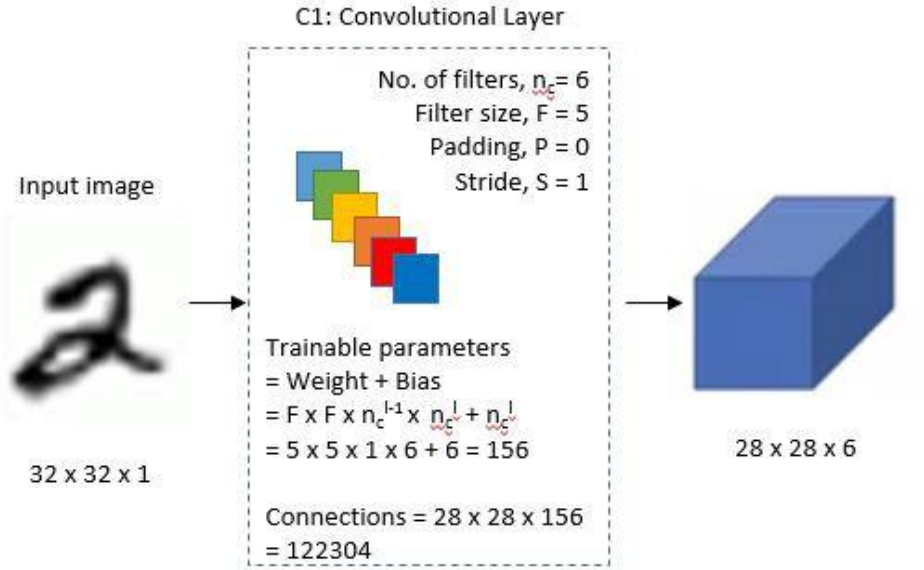
$$\text{input image} \rightarrow \text{gradients} + \text{edges} \rightarrow \text{textures} \rightarrow \text{patterns} \rightarrow \text{basic object features} \rightarrow \text{object} \rightarrow \text{output}$$

LeNet-5

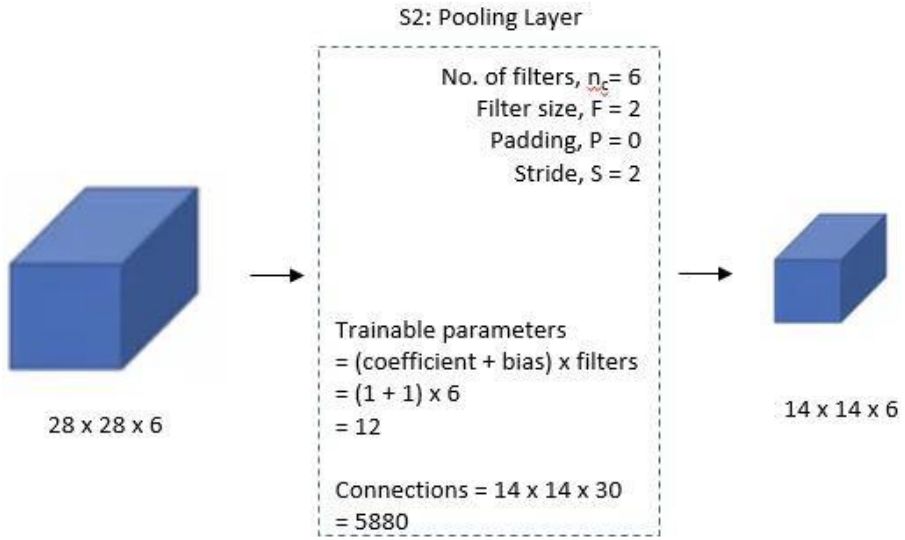

INPUT
32x32
C1: feature maps
6@28x28
S2: f. maps
6@14x14
C3: f. maps 16@10x10
S4: f. maps 16@5x5
C5: layer
120
F6: layer
84
OUTPUT
10
Convolutions
Subsampling
Convolutions
Subsampling
Full connection
Full connection
Gaussian connections

First Layer

The input for LeNet-5 is a 32×32 grayscale image which passes through the first convolutional layer with 6 feature maps or filters having size 5×5 and a stride of one. The image dimensions change from 32x32x1 to 28x28x6.



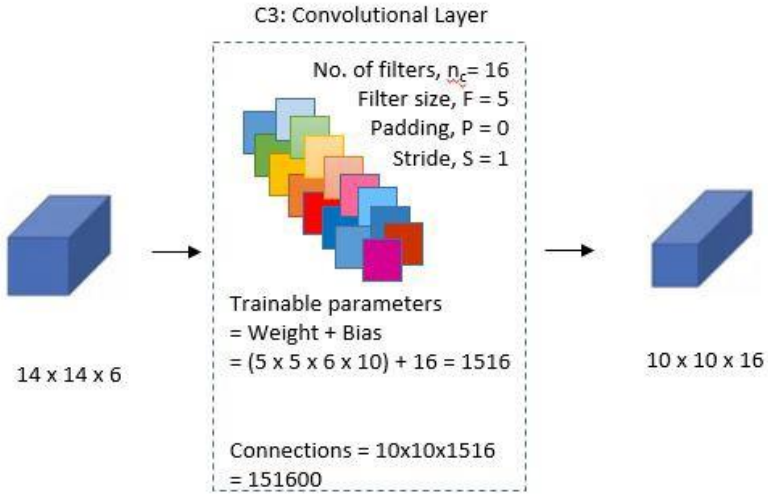
Second Layer



Then the LeNet-5 applies average pooling layer or sub-sampling layer with a filter size 2×2 and a stride of two. The resulting image dimensions will be reduced to 14x14x6.

Third Layer

Next, there is a second convolutional layer with 16 feature maps having size 5×5 and a stride of 1. In this layer, only 10 out of 16 feature maps are connected to 6 feature maps of the previous layer as shown below. The main reason is to break the symmetry in the network and keeps the number of connections within reasonable bounds. That's why the number of training parameters in this layers are 1516 instead of 2400 and similarly, the number of connections are 151600 instead of 240000.

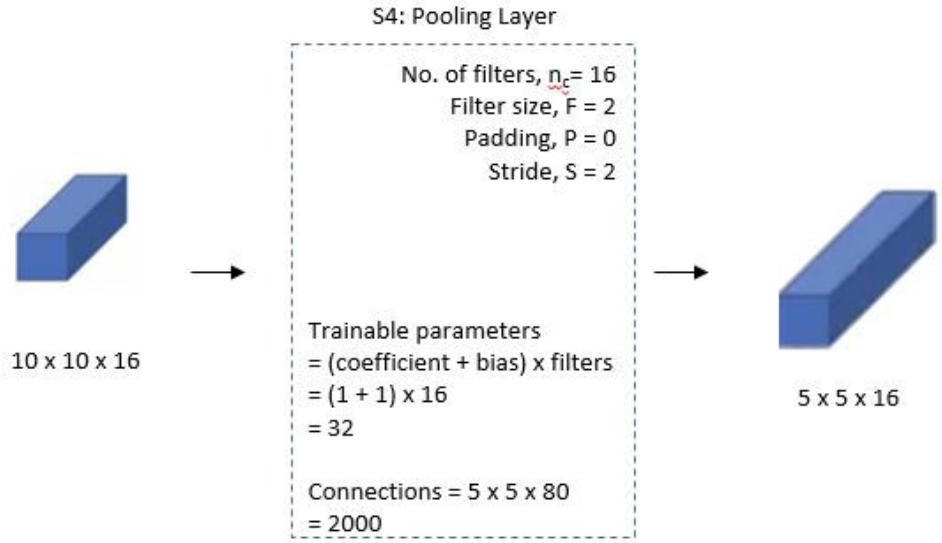


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I

EACH COLUMN INDICATES WHICH FEATURE MAP IN S2 ARE COMBINED BY THE UNITS IN A PARTICULAR FEATURE MAP OF C3.

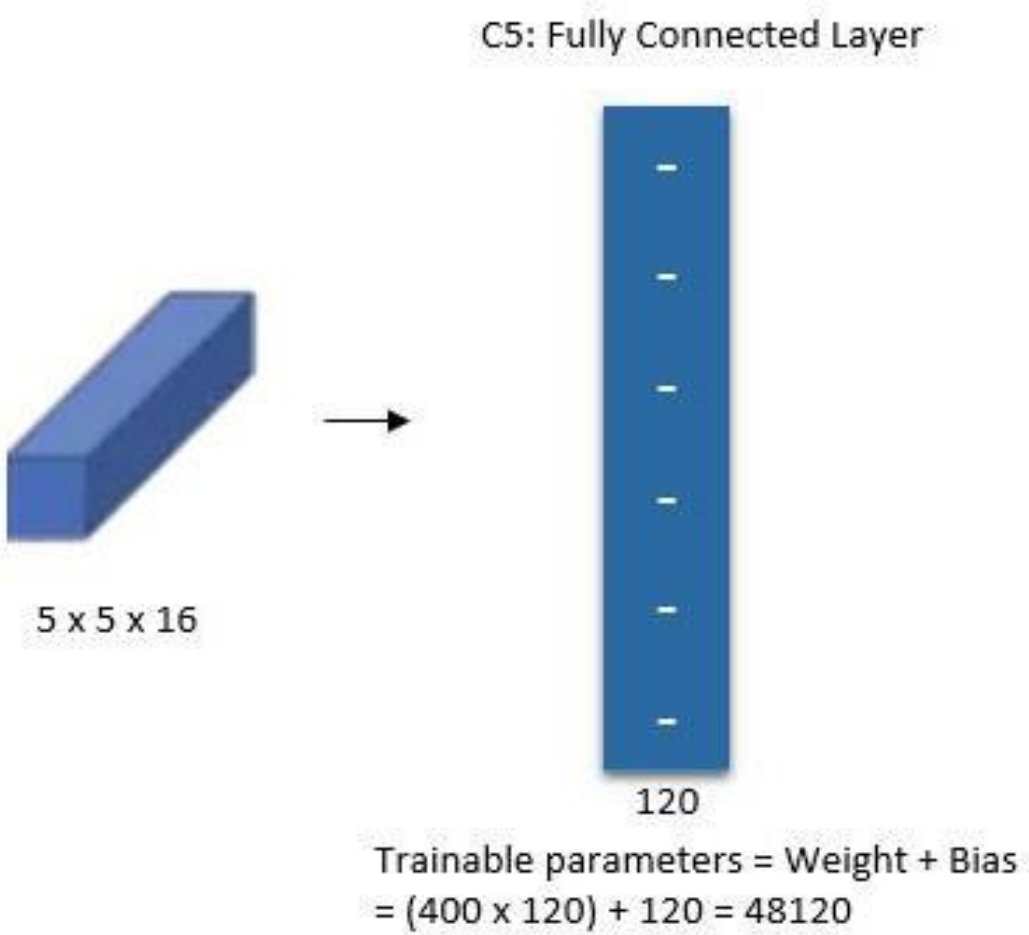
Fourth Layer



The fourth layer (S4) is again an average pooling layer with filter size 2×2 and a stride of 2. This layer is the same as the second layer (S2) except it has 16 feature maps so the output will be reduced to 5x5x16.

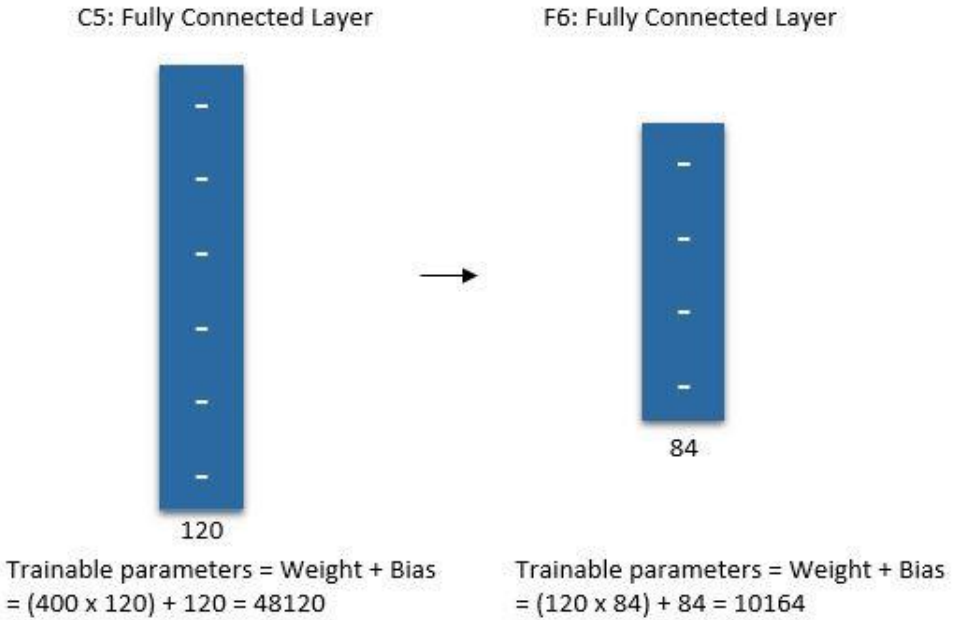
Fifth Layer

The fifth layer (C5) is a fully connected convolutional layer with 120 feature maps each of size 1×1. Each of the 120 units in C5 is connected to all the 400 nodes (5x5x16) in the fourth layer S4.



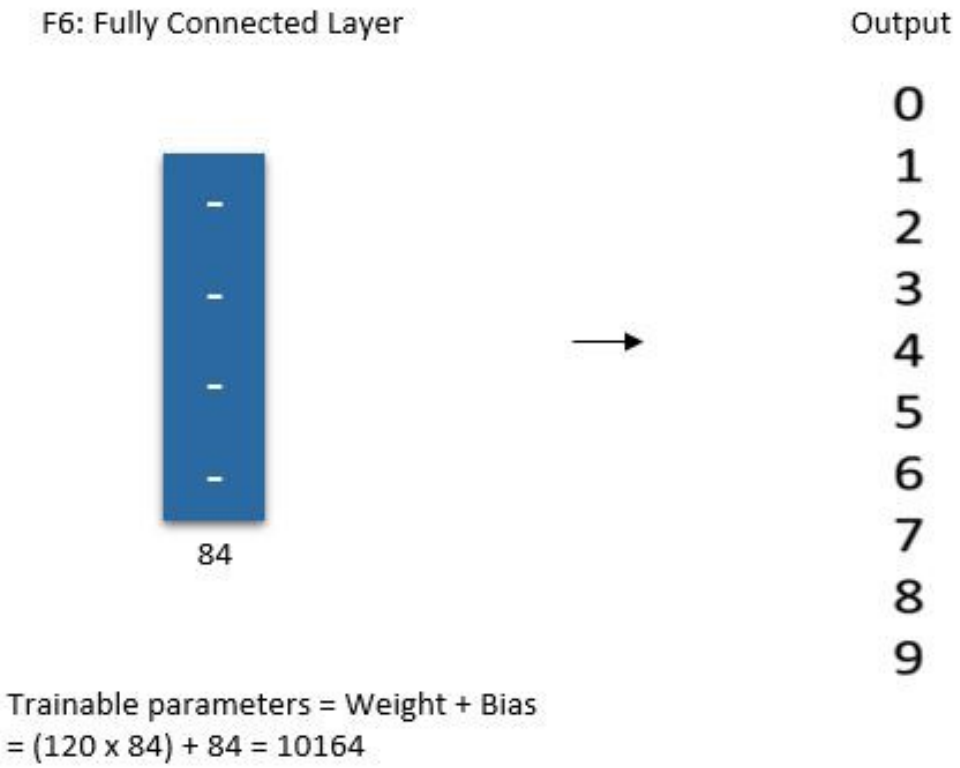
Sixth Layer

The sixth layer is a fully connected layer (F6) with 84 units .



Output Layer

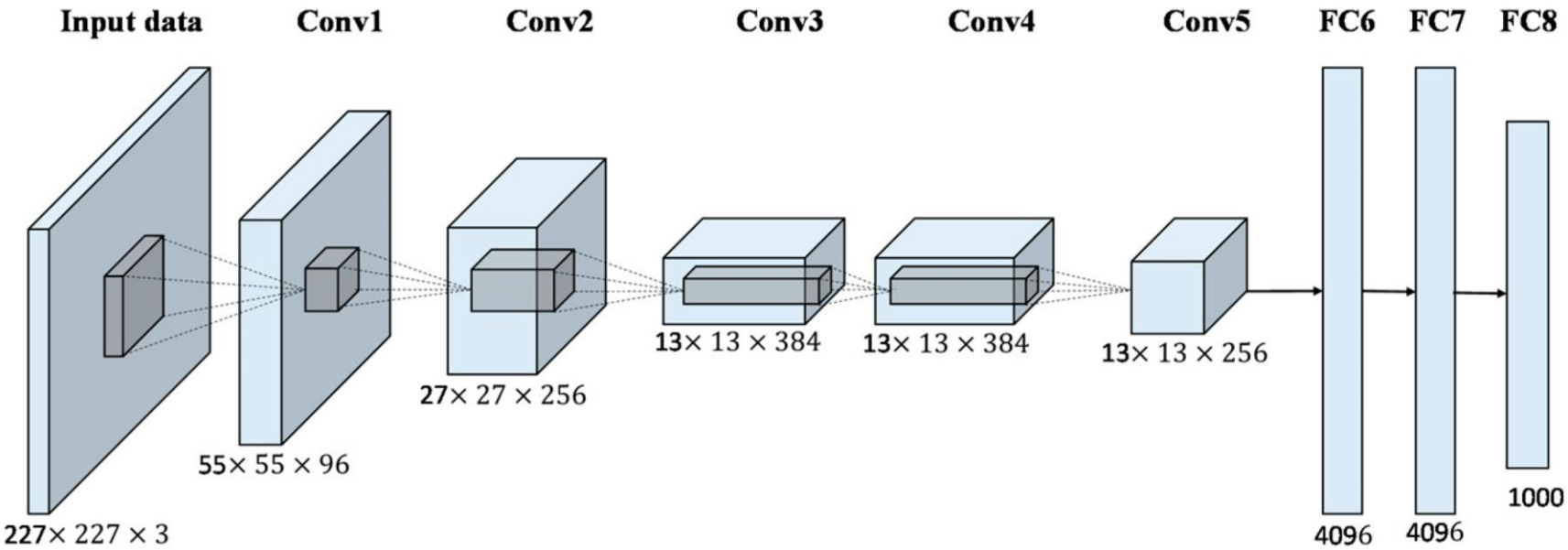
Finally, there is a fully connected **softmax** output layer ŷ with 10 possible values corresponding to the digits from 0 to 9.



Alexnet

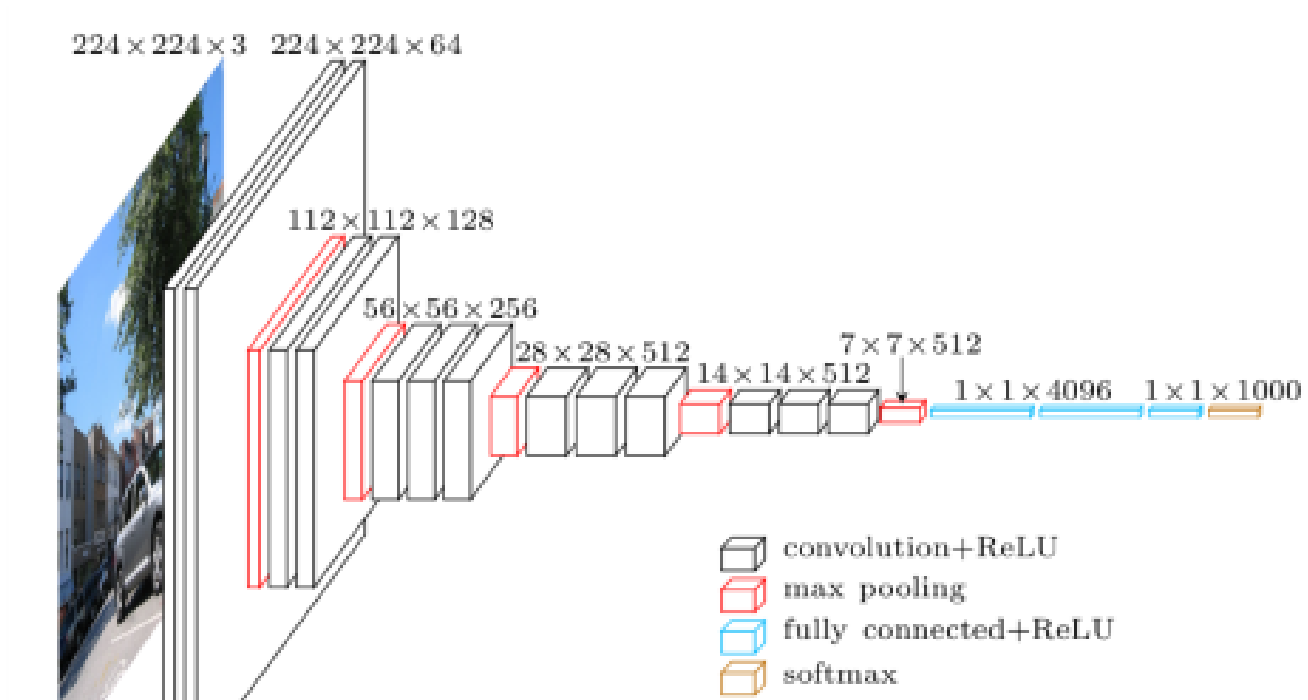
The **Alexnet** has eight layers with learnable parameters. The model consists of five layers with a combination of max pooling followed by 3 fully connected layers and they use Relu activation in each of these layers except the output layer.

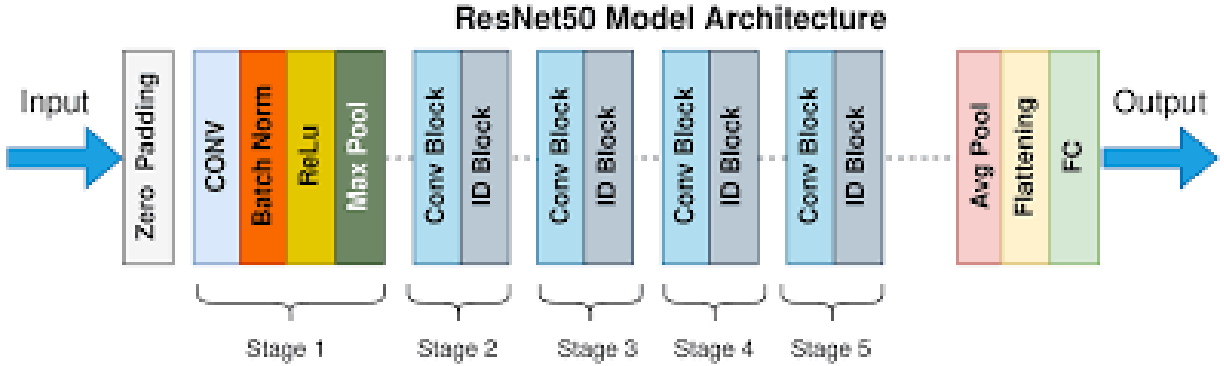
They found out that using the relu as an activation function accelerated the speed of the training process by almost six times. They also used the dropout layers, that prevented their model from overfitting. Further, the model is trained on the Imagenet dataset. The Imagenet dataset has almost 14 million images across a thousand classes.


Input data
Conv1
Conv2
Conv3
Conv4
Conv5
FC6
FC7
FC8
227× 227 × 3
55× 55 × 96
27× 27 × 256
13× 13 × 384
13× 13 × 384
13× 13 × 256
4096
4096
1000

VGG

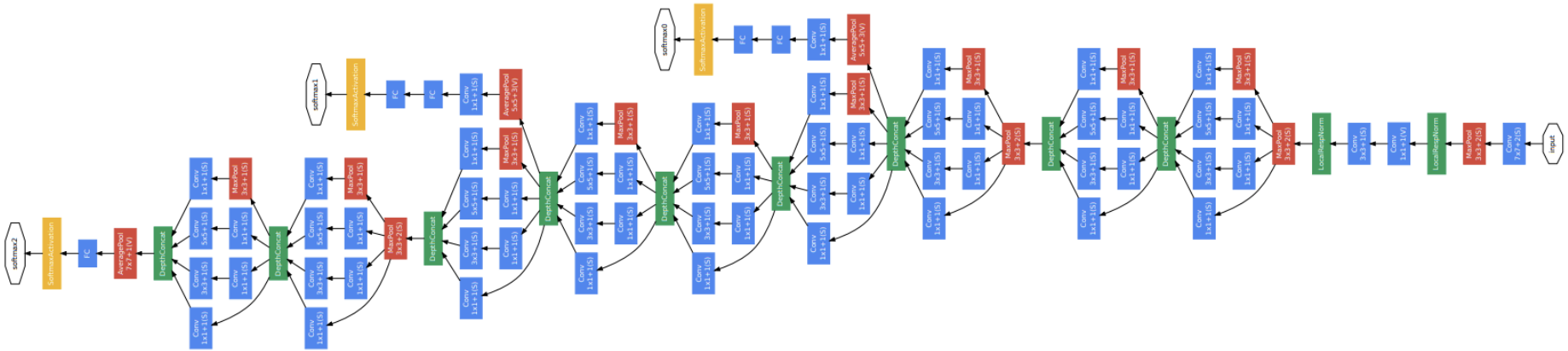
VGG stands for Visual Geometry Group; it is a standard deep CNN architecture with multiple layers. The “deep” refers to the number of layers with VGG-16 or VGG-19 consisting of 16 and 19 convolutional layers. The VGG architecture is the basis of ground-breaking object recognition models. Developed as a deep neural network, the VGGNet also surpasses baselines on many tasks and datasets beyond ImageNet.


224 × 224 × 3
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
convolution+ReLU
max pooling
fully connected+ReLU
softmax



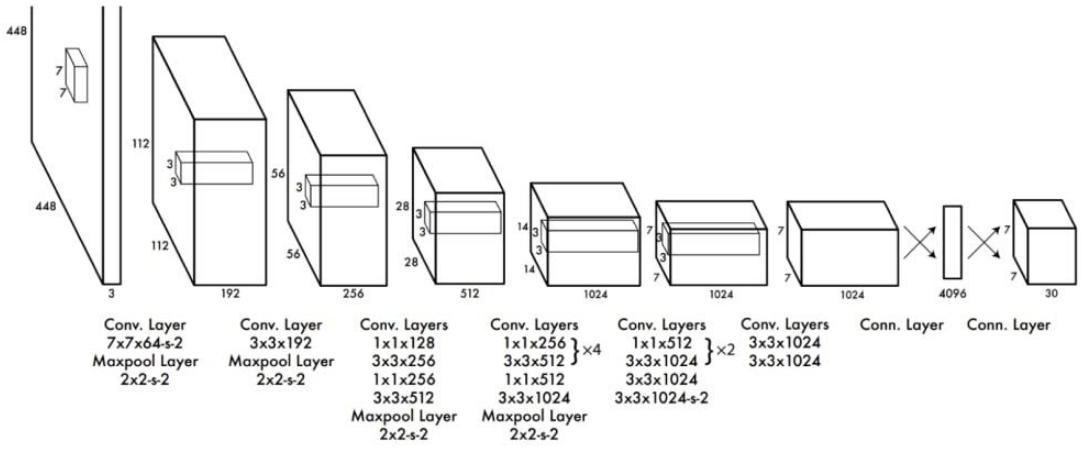
ResNet

ResNet (short for Residual Network) is a type of CNN architecture that was developed by Microsoft Research in 2015. ResNet is known for its deep layers and the use of residual connections, which allow for easier training of very deep networks by addressing the problem of vanishing gradients. Residual connections involve skipping one or more layers in a neural network and adding the output of an earlier layer to the output of a later layer. This allows the network to learn residual functions or differences between the input and output of a given layer, making it easier to optimize the network. ResNet has been widely used in various computer vision tasks and has achieved state-of-the-art results in image recognition, object detection, and image segmentation. Its architecture has also inspired many other deep learning models in the field.



GoogleNet

GoogLeNet, also known as Inception Net, is a CNN developed by researchers at Google. It is a 22-layer deep architecture and was trained on the ImageNet. ImageNet is a large-scale dataset that contains millions of labeled images spanning thousands of different categories for data collection in computer vision tasks. It can classify objects into 1,000 different categories. The GoogLeNet architecture is based on building a deeper model to achieve greater accuracy while keeping it computationally efficient. Neural networks with deeper architectures can capture complex patterns and extract hierarchical features which helps in generalizing better to new, unseen data.



The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

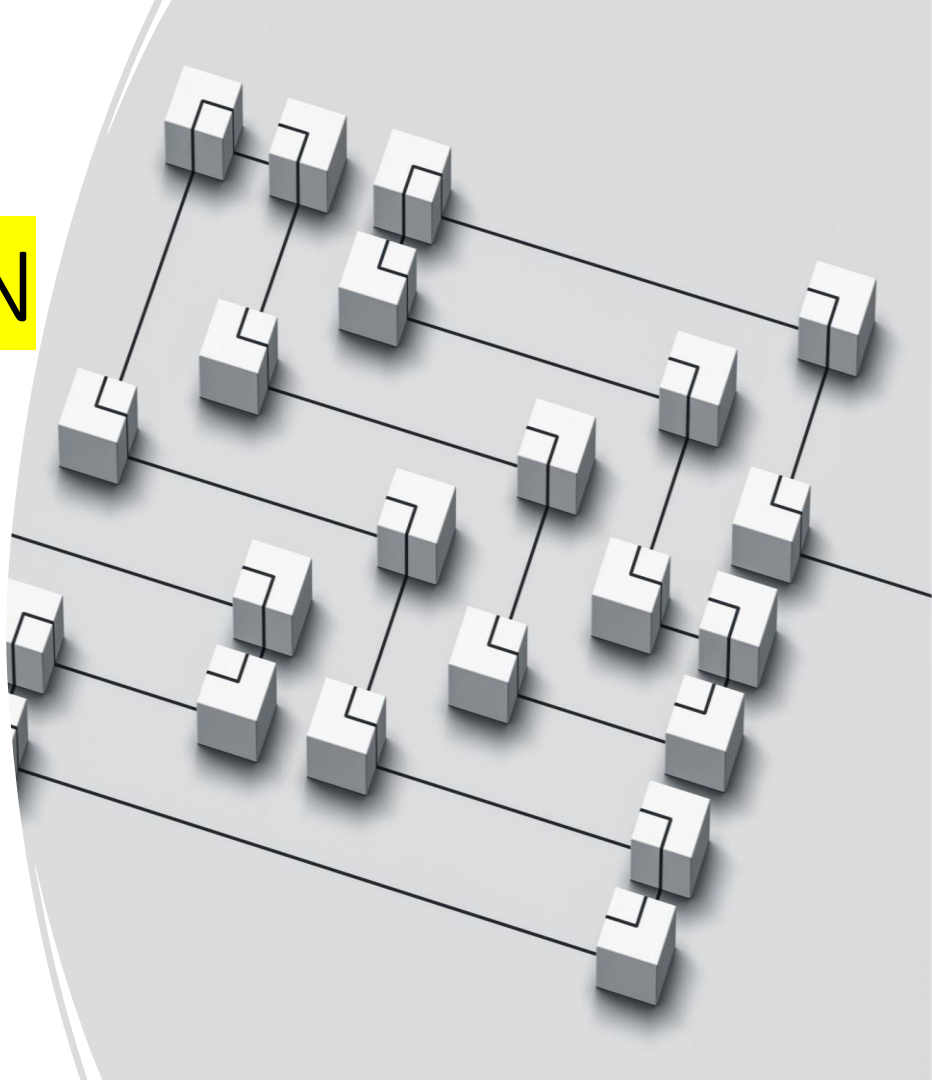
YOLO

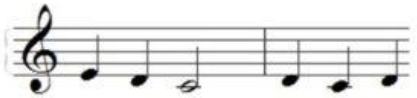
YOLO is a popular object detection model used in CNNs. It is known for its real-time detection capabilities and high accuracy in identifying various objects in images or videos. YOLO divides an image into grids and predicts bounding boxes and classes for each grid cell in a single forward pass through the network. This allows for faster and more efficient object detection compared to traditional methods that require multiple passes through the image. YOLO has been widely adopted in various computer vision tasks and has inspired several improvements and variations, such as YOLOv2, YOLOv3, and YOLOv4, which further enhance the performance and versatility of the original YOLO model.

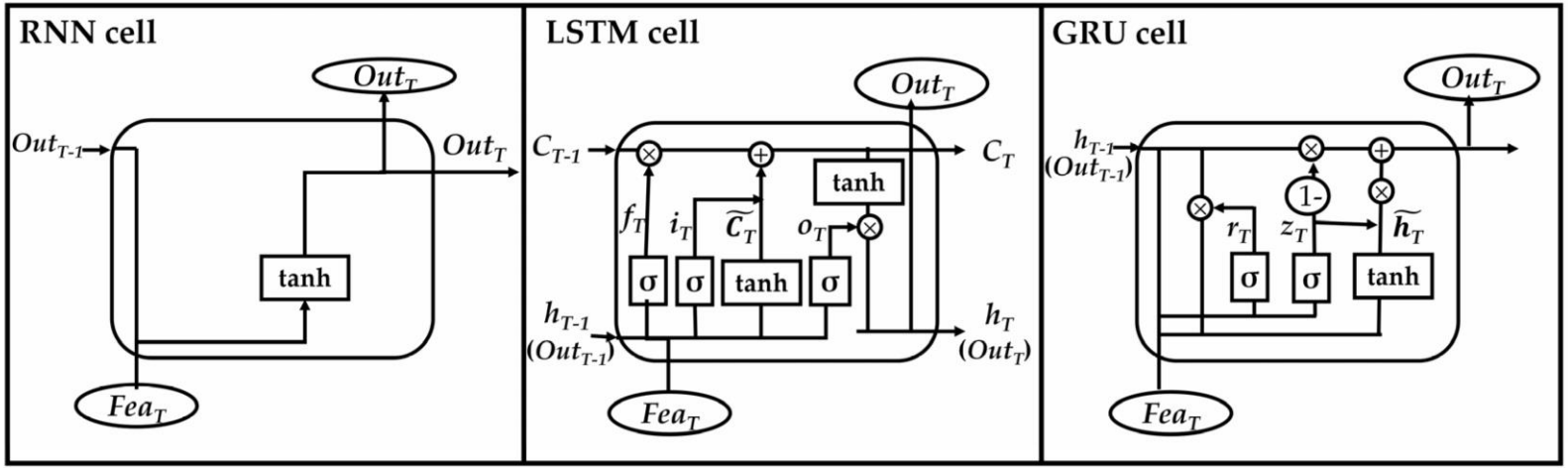
Sequence models-RNN

Sequences consist of data points that can be meaningfully ordered such that observations at one location in the sequence provide useful information about observations at other location(s). Countless supervised learning tasks require dealing with sequence data and depending on the nature of the input and output, one of the following three situations can arise in a sequence learning problem:

- The input is a sequence, while the target output is a single data point – Examples include video activity recognition, sentiment classification, stock price prediction, etc.
- 2. The input is a single data point, while the target output is a sequence – Examples include image captioning, music generation, speech synthesis, etc.
- 3. Both the input and the target output are sequences – Examples include speech recognition, natural language translation, DNA sequence analysis, name entity recognition, etc. It is worth mentioning that the input and output sequences may be of the same or different lengths.



	x - input		y - output
Speech recognition		⟹	"Fuzzy Wuzzy was a bear. Fuzzy Wuzzy had no hair."
Music generation	∅	⟹	
Sentiment classification	"Decent effort. The plot could have been better."	⟹	
DNA sequence analysis	ACTGTACCCATGTGACTGCCC	⟹	ACTGTACCCATGTGACTGCCC
Machine translation	"El que no arriesga, no gana."	⟹	"If you don't take risks, you cannot win."
Video activity recognition		⟹	Running
Name entity recognition	"Ygritte says Jon Snow knows nothing."	⟹	"Ygritte says Jon Snow knows nothing."

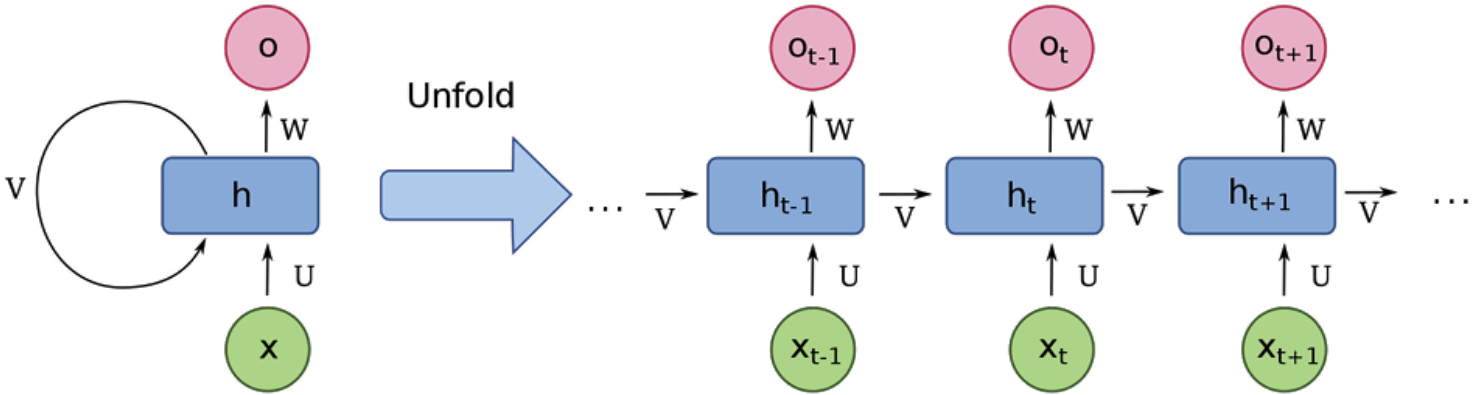

RNN cell
Out_T
Out_{T-1}
Out_T
tanh
Fea_T
LSTM cell
Out_T
C_{T-1}
C_T
tanh
f_T
i_T
$\widetilde{C}_T$
o_T
σ
σ
tanh
σ
h_{T-1}
(Out_{T-1})
h_T
(Out_T)
Fea_T
GRU cell
Out_T
h_{T-1}
(Out_{T-1})
1-
r_T
z_T
$\widetilde{h}_T$
σ
σ
tanh
Fea_T

(a) (b) (c)

Recurrent Neural Network (RNN)

- A recurrent neural network is an artificial neural network commonly used in speech recognition and natural language processing. RNNs recognize data's sequential characteristics and use patterns to predict the next likely scenario.
- RNNs are used in deep learning and in developing models that simulate neuron activity in the human brain. They are especially powerful in use cases where context is critical to predicting an outcome and are also distinct from other types of artificial neural networks because they use feedback loops to process a sequence of data that informs the final output. These feedback loops allow information to persist. **This effect is often described as memory.**
- RNN use cases tend to be connected to language models in which knowing the next letter in a word or the next word in a sentence is predicated on the data that comes before it. Writing by RNNs is a form of computational creativity. This simulation of human creativity is made possible by the AI's understanding of grammar and semantics learned from its training set.

Recurrent neural network



RNN

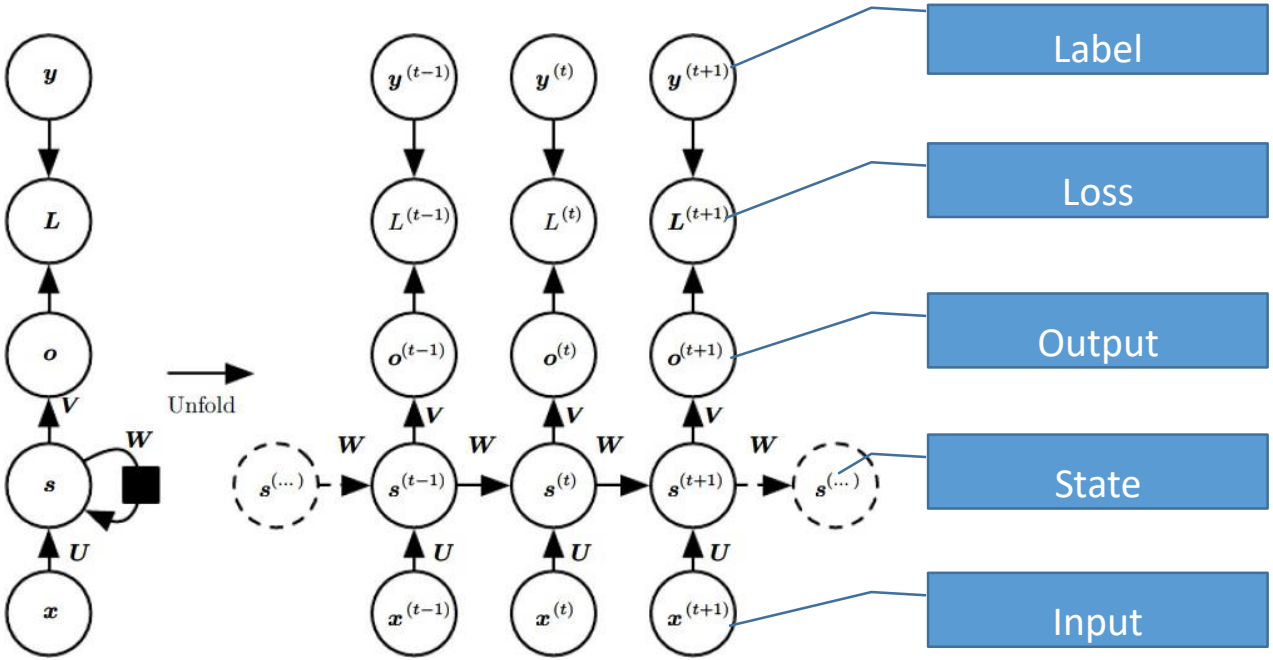
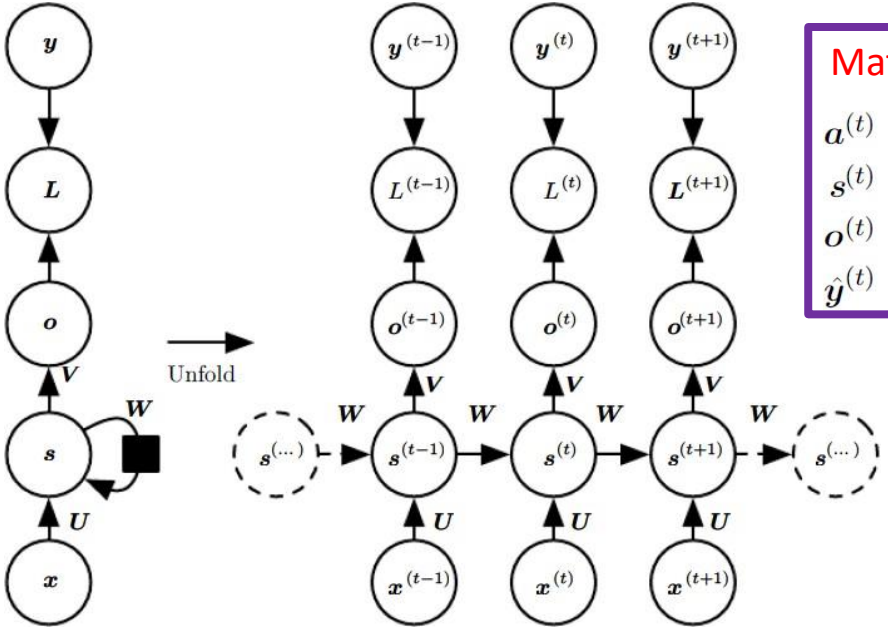


Figure from *Deep Learning*, by Goodfellow, Bengio and Courville

RNN

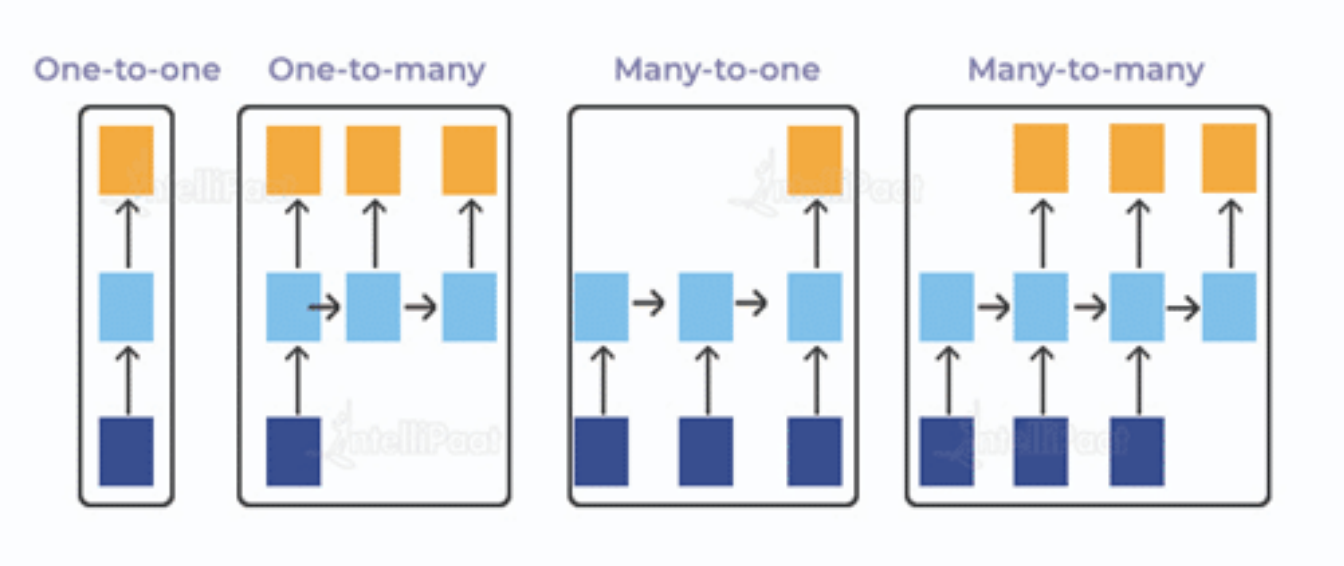


Math formula:

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{s}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

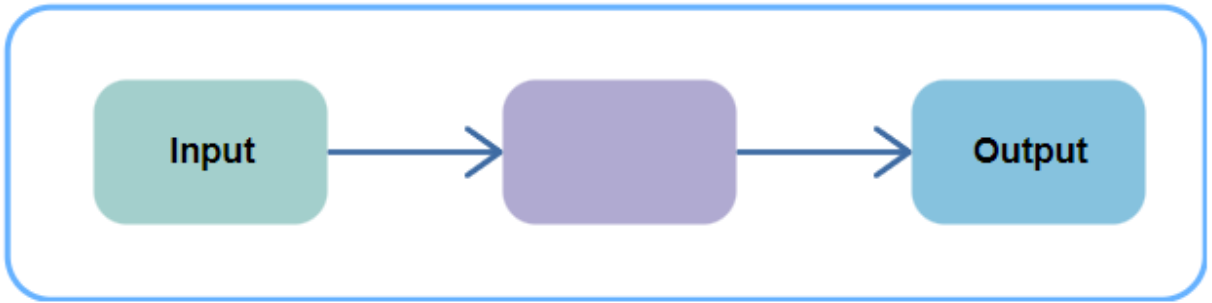
Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Type of RNN



One to one

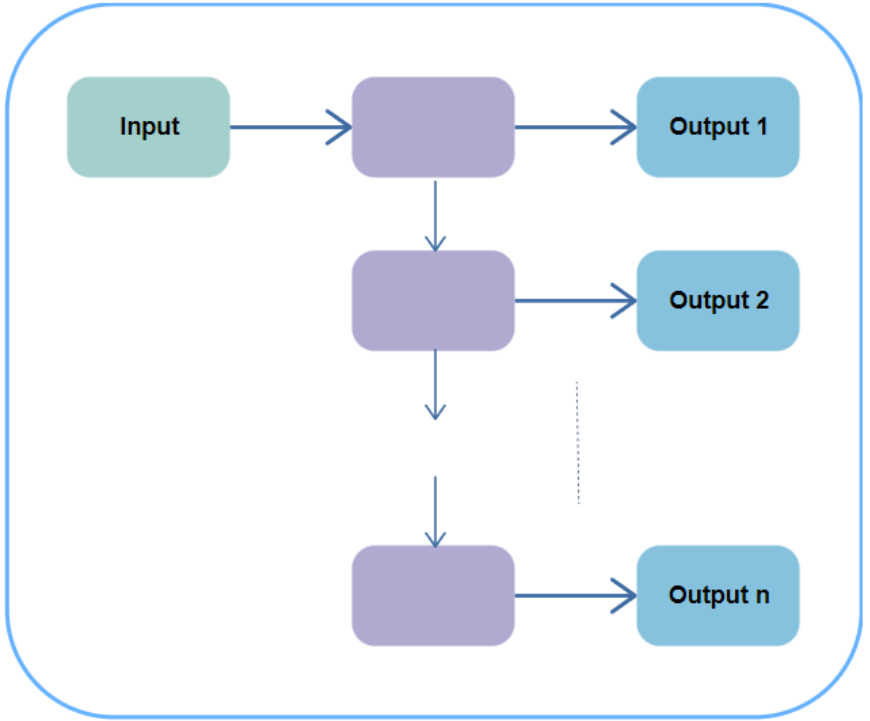
The simplest type of RNN is One-to-One, which allows a single input and a single output. It has fixed input and output sizes and acts as a traditional neural network. The One-to-One application can be found in Image Classification .



One-to-One

One-to-Many

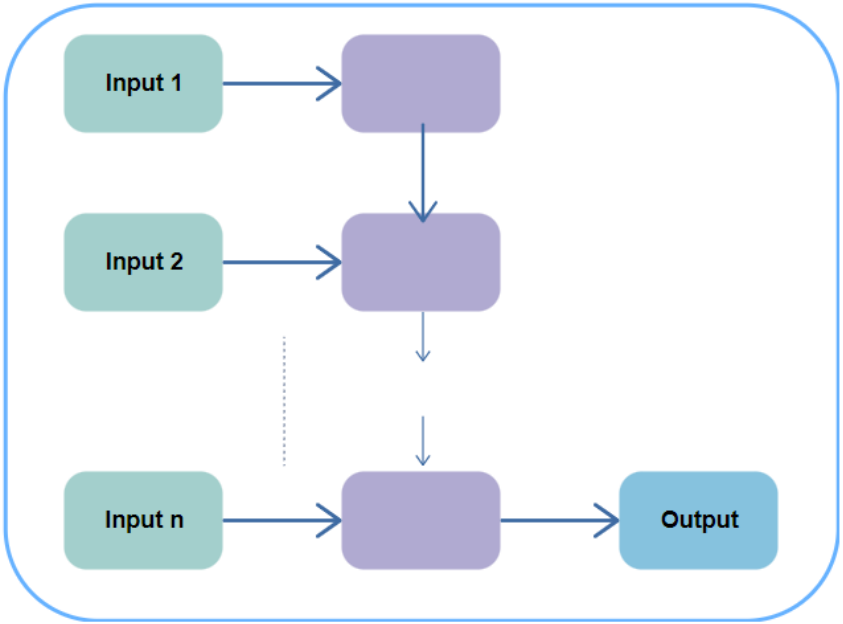
One-to-Many is a type of RNN that gives multiple outputs when given a single input. It takes a fixed input size and gives a sequence of data outputs. Its applications can be found in Music Generation and Image Captioning.



One-to-Many

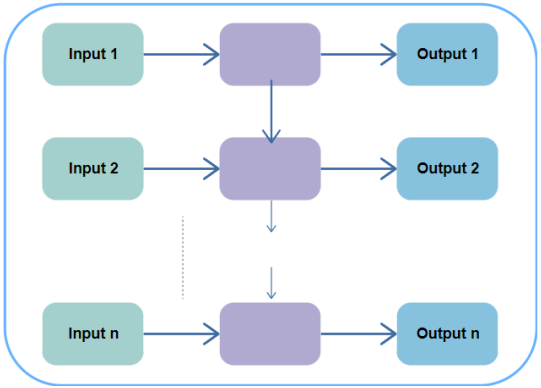
Many-to-One

Many-to-One is used when a single output is required from multiple input units or a sequence of them. It takes a sequence of inputs to display a fixed output. Sentiment Analysis is a common example of this type of RNN.



Many-to-One

Many-to-Many

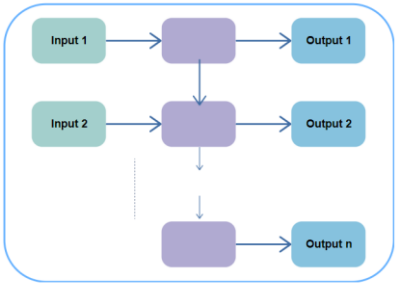


Many-to-Many (Equal)

Many-to-Many generates a sequence of output data from a sequence of input units.

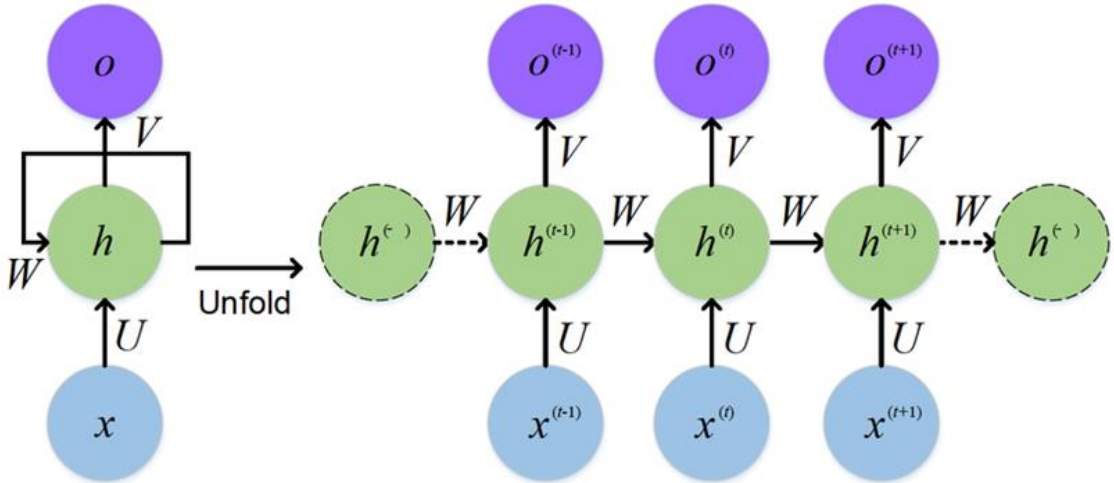
This type of RNN is further divided into the following two subcategories:

- Equal Unit Size: In this case, the number of both the input and output units is the same. A common application can be found in Name-Entity Recognition.
- Unequal Unit Size: In this case, inputs and outputs have different numbers of units. Its application can be found in Machine Translation.

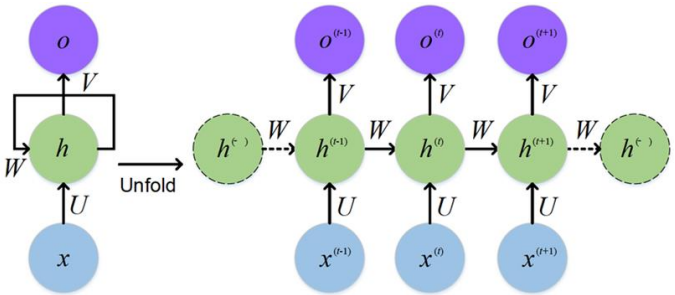


Many-to-Many (Unequal)

Unfolding a Recurrent Neural Network

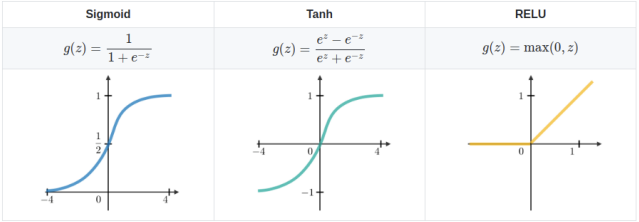


Components



- **Input Vector (x)**: Represented by the blue circles, these are the inputs at each time step.
- **Hidden State (h)**: Represented by the green circles, these are the internal states of the RNN that capture information from previous time steps.
- **Output Vector (o)**: Represented by the purple circles, these are the outputs at each time step.
- **Weight Matrices**:
  - U: The weight matrix connecting the input vector x to the hidden state h.
  - W: The weight matrix connecting the hidden state h from the previous time step to the current hidden state.
  - V: The weight matrix connecting the hidden state h to the output vector o.

Process



Initial State: The RNN starts with an initial hidden state h, which is often initialized to zero or some other value.

Unfolding Through Time: The RNN is unfolded through time to process sequences. Each time step t involves the following computations :

- **Hidden State Update**: The hidden state at time $t\left(h^{(t)}\right)$ is computed using the input at time $t(x^t)$ and the hidden state from the previous time step (h^{t-1}). This is done using the weight matrices U and W:

$$h(t) = \sigma(Ux(t) + Wh(t-1))$$

where σ is an activation function, typically a non-linear function like tanh or ReLU.

- **Output Computation**: The output at time $t(o^t)$ is computed using the current hidden state h^t and the weight matrix V:

$$o(t) = \phi(Vh(t))$$

where ϕ is an activation function, often a softmax function for classification tasks.

Hands-on Example

Suppose we have a sequence of numbers: [2, 4, 6, 8, 10].
We want to use an RNN to predict the next number in the sequence.
We'll create a simple RNN with a single hidden unit.

- We'll use a simple RNN with the following equations:
  - $h(t)=\sigma(Ux(t)+Wh(t-1))$
  - $o(t)=\phi(Vh(t))$
- Let's initialize the weight matrices as follows:
  - $U=1$, $W=1$, $V=1$
  - Let's use the ReLU activation function for the hidden state and output (σ and ϕ) .

$for\ x^1 = 2:$
$Hidden\ state \Rightarrow h^1 = ReLU(1 \times 2 + 1 \times 0) = ReLU(2) = 2$
$Output \Rightarrow o^1 = ReLU(1 \times 2) = ReLU(2) = 2$

$for\ x^2 = 4:$
$Hidden\ state \Rightarrow h^2 = ReLU(1 \times 4 + 1 \times 2) = ReLU(6) = 6$
$Output \Rightarrow o^2 = ReLU(1 \times 6) = ReLU(6) = 6$

$for\ x^3 = 6 \Rightarrow h^3 = 10\ , o^3 = 10$

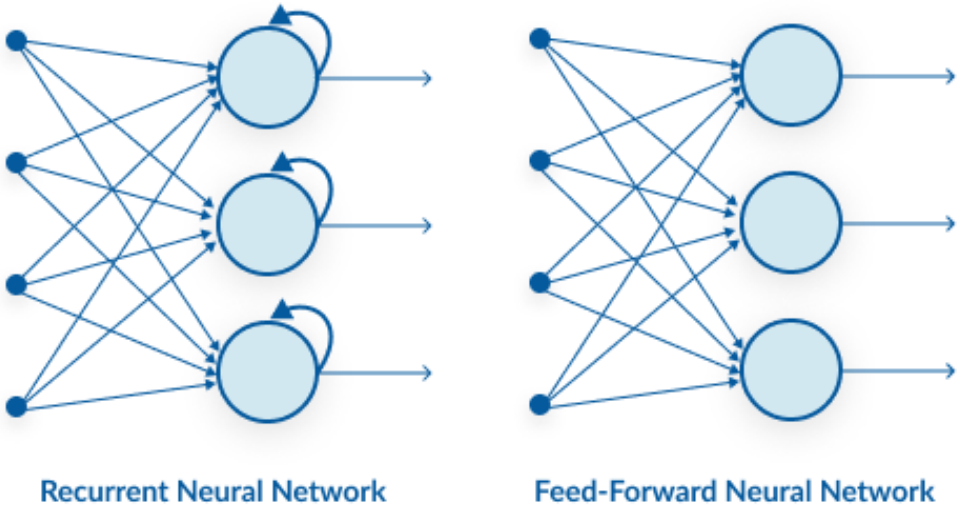
$for\ x^4 = 8 \Rightarrow h^4 = 14\ , o^4 = 14$

$for\ x^5 = 10 \Rightarrow h^5 = 18\ , o^5 = 18$

- We'll use a simple RNN with the following equations:
  - $h(t)=\sigma(Ux(t)+Wh(t-1))$
  - $o(t)=\phi(Vh(t))$
- Let's initialize the weight matrices as follows:
  - $U=1\ , W=1\ , V=1$
  - Let's use the ReLU activation function for the hidden state and output (σ and ϕ) .

In this example, the RNN successfully predicted the next number in the sequence [2, 4, 6, 8, 10] to be 18 based on the pattern it learned from the input sequence.
This demonstrates the basic idea of how RNNs can capture temporal dependencies in sequential data to make predictions.

Recurrent Neural Network vs Feedforward Neural Network



FNNs: A feed-forward neural network has only one route of information flow: from the input layer to the output layer, passing through the hidden layers. The data flows across the network in a straight route, never going through the same node twice.
A feed-forward neural network can perform simple classification, regression, or recognition tasks, but it can't remember the previous input that it has processed that is why FNNs are poor predictions of what will happen next because they have no memory of the information they receive.
Because it simply analyses the current input, a feed-forward network has no idea of temporal order. Apart from its training, it has no memory of what transpired in the past.

RNNs: The information is in an RNN cycle via a loop. Before making a judgment, it evaluates the current input as well as what it has learned from past inputs. A recurrent neural network, on the other hand, may recall due to internal memory. It produces output, copies it, and then returns it to the network.

FNNs: Typically work best **with fixed-length inputs** and outputs. They excel at pattern recognition tasks where the data points are independent of each other. For instance, image recognition or spam email classification.

RNNs: **sequential data**, where the order and relationships between elements matter. This makes them ideal for tasks like speech recognition, machine translation, and text generation where the meaning unfolds over time.

Application

FNNs: Power applications like image recognition, medical diagnosis (analyzing X-rays to detect abnormalities), image classification and spam filtering (identifying unwanted emails).

RNNs: Drive tasks like speech recognition (understanding spoken language), machine translation (converting text from one language to another), text generation (creating chatbots or writing different content formats), and time series forecasting (predicting stock prices or weather patterns).

Backpropagation Through Time (BPTT)

RNN uses a technique called Backpropagation through time to backpropagate through the network to adjust their weights so that we can reduce the error in the network. It got its name “through time” as in RNN we deal with sequential data and every time we go back it’s like going back in time towards the past. In the BPTT step, we calculate the partial derivative at each weight in the network. So if we are in time t = 3, then we consider the derivative of E3 with respect to that of S3. Now, x3 is also connected to s3. So, its derivative is also considered. Now if we see s3 is connected to s2 so s3 is depending on the value from s2 and here derivative of s3 with respect to s2 is also considered. This acts as a chain rule and we accumulate all the dependency with their derivatives and use it for error calculation.

In E3 we have a gradient that is from S3 and its equation at that time is:

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial W_s}$$

Now we also have s2 associated with s3 so :

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial \bar{s}_2}\frac{\partial \bar{s}_2}{\partial W_s}$$

And s1 is also associated with s2 and hence now all s1,s2,s3 and having an effect on E3

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial \bar{s}_2}\frac{\partial \bar{s}_2}{\partial \bar{s}_1}\frac{\partial \bar{s}_1}{\partial W_s}$$

On accumulating everything, we end up getting this equation that Ws has contributed towards that network at time t=3

$$\begin{aligned}\frac{\partial E_3}{\partial W_s} = &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial W_s} + \\ &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial \bar{s}_2}\frac{\partial \bar{s}_2}{\partial W_s} + \\ &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial \bar{s}_2}\frac{\partial \bar{s}_2}{\partial \bar{s}_1}\frac{\partial \bar{s}_1}{\partial W_s}\end{aligned}$$

The general equation for which we adjust Ws in our BPTT network can be written as

$$\frac{\partial E_N}{\partial W_s} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W_s}$$

Now as we have noticed Wx is also associated with the network. So, doing the same we can generally write

$$\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W_x}$$

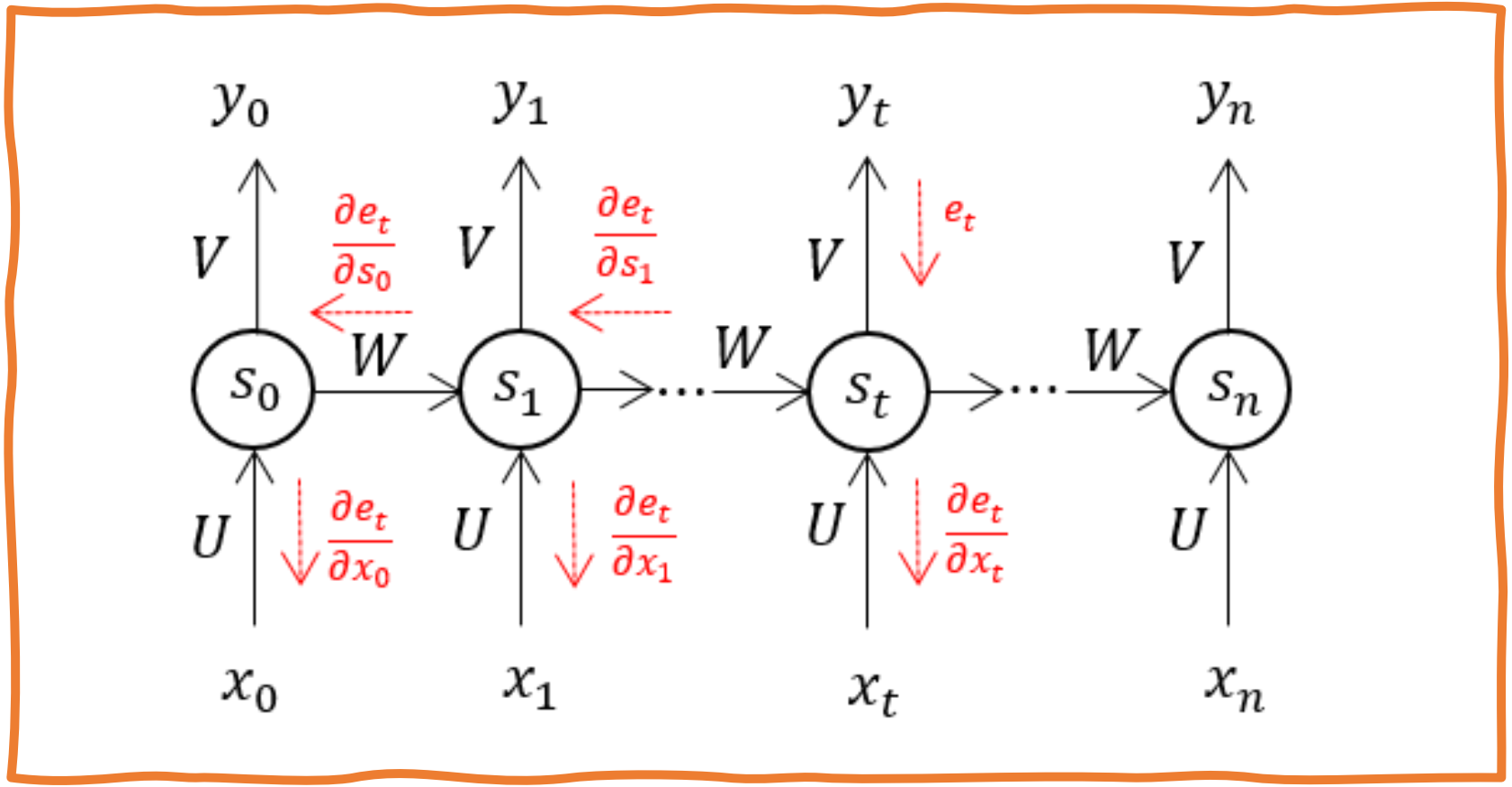
Now that you have understood how BPTT works, this is basically all about how RNN adjusts its weights and reduces the error. The main fault here is that this is basically only for a small network with 4 layers. But imagine if we had hundreds of layers and at a time let's say t = 100, we would end up calculating all the partial derivatives associated with the network and this is a huge multiplication and this can bring down the overall value to a very small or minute value such that it may end up being useless to correct the error. This issue is called **Vanishing Gradient Problem**.

Vanishing Gradient Problem

As we all know that in RNN to predict an output we will be using a sigmoid activation function so that we can get the probability output for a particular class. As we saw in the above section when we are dealing with say E3 there is a long-term dependency. The issue occurs when we are taking the derivative and derivative of the sigmoid is always below 0.25 and hence when we multiply a lot of derivatives together according to the chain rule, we end up with a vanishing value such that we can't use them for error calculation. Thus the weights and biases won't get updated properly and as layers keep increasing we fell more into this and our model doesn't work properly and leads to inaccuracy in the entire network. Some ways to solve this problem are to either initialize the weight matrix properly or go for something like a ReLU instead of sigmoid or tanh functions.

Exploding Gradient Problem

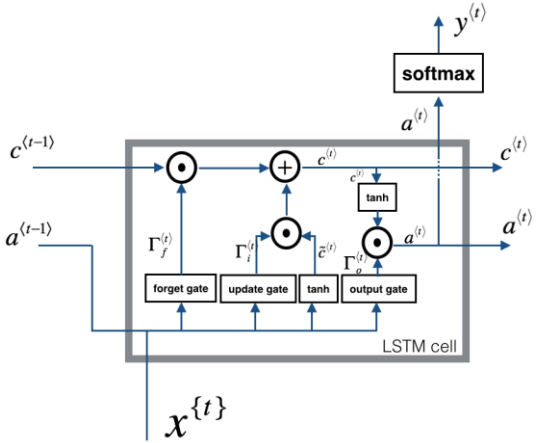
Exploding gradients is a problem in which the gradient value becomes very big and this often occurs when we initialize larger weights and we could end up with NaN. If our model suffers from this issue, we can't update the weights at all. But luckily, gradient clipping is a process that we can use for this. At a pre-defined threshold value, we clip the gradient. This will prevent the gradient value from going beyond the threshold, and we will never end up in big numbers or NaN.


y_0
y_1
y_t
y_n
V
$\frac{\partial e_t}{\partial s_0}$
$\frac{\partial e_t}{\partial s_1}$
e_t
W
s_0
s_1
s_t
s_n
U
$\frac{\partial e_t}{\partial x_0}$
$\frac{\partial e_t}{\partial x_1}$
$\frac{\partial e_t}{\partial x_t}$
x_0
x_1
x_t
x_n

LSTM

One way to solve the problem of Vanishing gradient and Long term dependency in RNN is to go for LSTM networks. Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work. They work tremendously well on a large variety of problems, and are now widely used. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn. All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

LSTM , Step by Step



$$\Gamma_f^{\langle t \rangle} = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}$$
$$\Gamma_u^{\langle t \rangle} = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}$$
$$\tilde{c}^{\{t\}} = \tanh(W_C[a^{\langle t-1 \rangle}, x$$
$$c^{\langle t \rangle} = \Gamma_f^{\langle t \rangle} \circ c^{\langle t-1 \rangle} + \Gamma_u^{\langle t \rangle}$$
$$\Gamma_o^{\langle t \rangle} = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}$$
$$a^{\langle t \rangle} = \Gamma_o^{\langle t \rangle} \circ \tanh(c^{\langle t \rangle})$$

Components and Notations

x^t: Input at time step tt.

a^{t-1}Hidden state from the previous time step.

c^{t-1}: Cell state from the previous time step.

a^t: Hidden state at the current time step.

c^t:Cell state at the current time step.

Γ_f^t: Forget gate.

$\Gamma_{\mathrm{u}}^{\mathrm{t}}$: Update (input) gate.

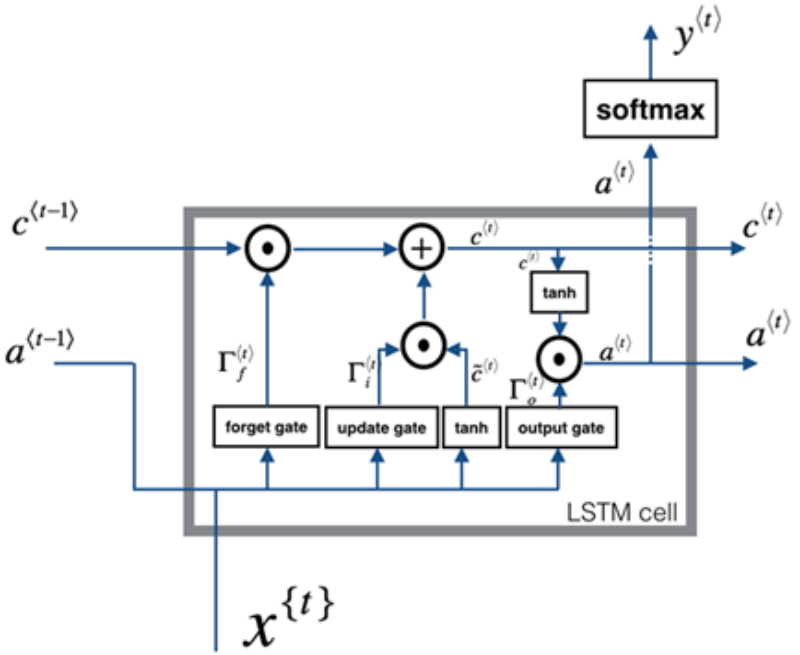
$c^{\sim t}$Candidate cell state.

Γ_o^t: Output gate.

W and b: Weights and biases for the respective gates.

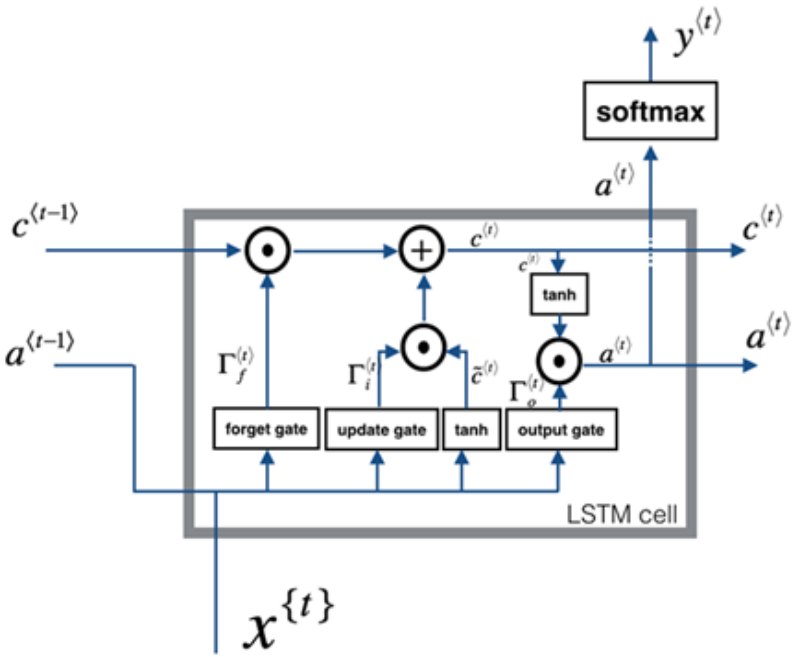
1. **Forget Gate ($\Gamma_f^{(t)}$)**

   - **Equation**: $\Gamma_f^{(t)} = \sigma(W_f[a^{(t-1)}, x^{(t)}] + b_f)$
   - **Function**: Decides what information to discard from the cell state.
   - **Operation**: Takes the previous hidden state $a^{(t-1)}$ and the current input $x^{(t)}$, applies a linear transformation followed by a sigmoid activation function σ.



2. **Update (Input) Gate ($\Gamma_u^{(t)}$)**

- **Equation**: $\Gamma_u^{(t)} = \sigma(W_u[a^{(t-1)}, x^{(t)}] + b_u)$
- **Function**: Decides what new information to add to the cell state.
- **Operation**: Similar to the forget gate, it processes $a^{(t-1)}$ and $x^{(t)}$ through a linear transformation and a sigmoid activation function.

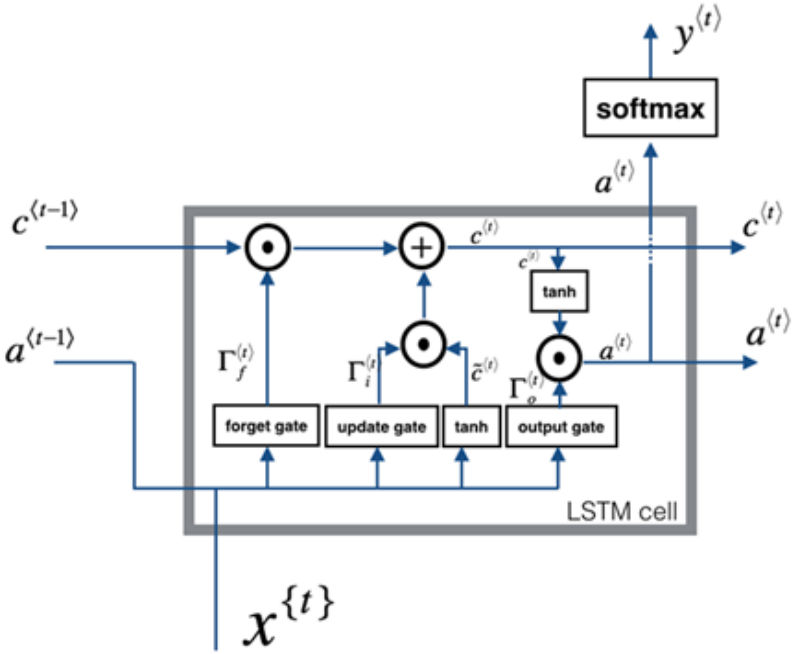


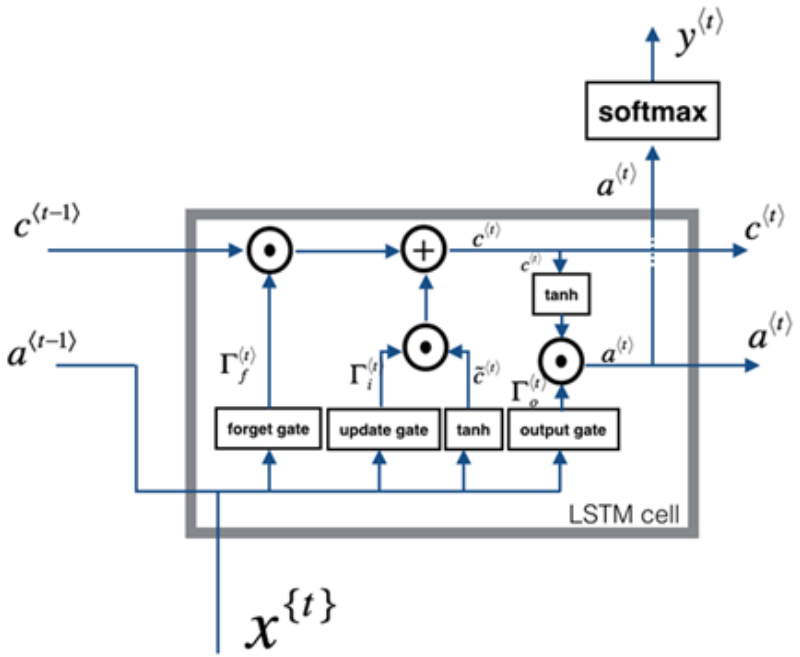
3. **Candidate Cell State ($\tilde{c}^{(t)}$)**

- **Equation**: $\tilde{c}^{(t)} = \tanh(W_c[a^{(t-1)}, x^{(t)}] + b_c)$
- **Function**: Creates a vector of new candidate values that could be added to the cell state.
- **Operation**: Uses a tanh activation function to ensure the values are between -1 and 1.

4. **Cell State Update ($c^{(t)}$)**

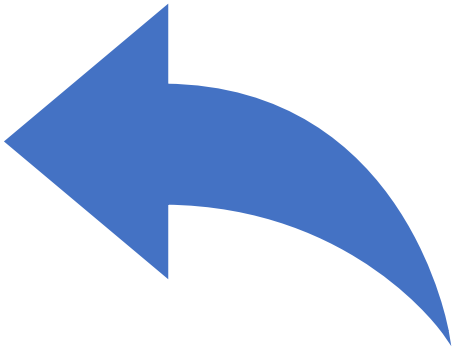
- **Equation**: $c^{(t)} = \Gamma_f^{(t)} \odot c^{(t-1)} + \Gamma_u^{(t)} \odot \tilde{c}^{(t)}$
- **Function**: Updates the cell state by combining the old cell state and the new candidate values.
- **Operation**: Element-wise multiplication (Hadamard product) is used to combine the forget gate's decision and the update gate's decision.





5. **Output Gate ($\Gamma_o^{(t)}$)**
   - **Equation**: $\Gamma_o^{(t)} = \sigma(W_o[a^{(t-1)}, x^{(t)}] + b_o)$
   - **Function**: Decides what part of the cell state to output.
   - **Operation**: Processes $a^{(t-1)}$ and $x^{(t)}$ through a linear transformation and a sigmoid activation function.

6. **Hidden State ($a^{(t)}$)**
   - **Equation**: $a^{(t)} = \Gamma_o^{(t)} \odot \tanh(c^{(t)})$
   - **Function**: The final output of the LSTM cell for the current time step.
   - **Operation**: Combines the output gate's decision with the updated cell state, passed through a tanh activation function.



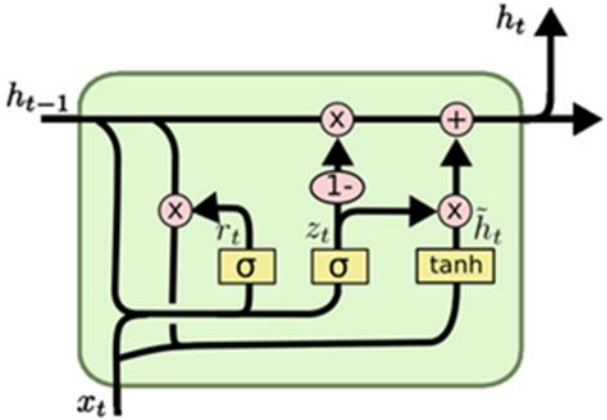
Summary

- The **forget gate** decides what information to discard from the previous cell state.
- The **update gate** decides what new information to add to the cell state.
- The **candidate cell** state is a potential new addition to the cell state.
- The **cell state** is updated by combining the old cell state and the new candidate values.
- The **output gate** decides what part of the cell state to output.
- The **hidden state** is the final output of the LSTM cell for the current time step.

GRU (Gated Recurrent Unit)

Gated Recurrent Unit (GRU) is an advancement of the standard RNN. It was introduced by Kyunghyun Cho et al. in 2014. GRUs are very similar to LSTM. Just like LSTM, GRU uses gates to control the flow of information. They are relatively new as compared to LSTM. This is why they offer some improvement over LSTM and have simpler architecture. Another Interesting thing about GRU network is that, unlike LSTM, it does not have a separate cell state . It only has a hidden state. Due to the simpler architecture, GRUs are faster to train.

GRU step by step



1. **Reset Gate (r_t):**
   - The reset gate determines how much of the previous hidden state (h_{t-1}) to forget.
   - It is calculated using the following formula:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

   - Here, σ is the sigmoid activation function, W_r is the weight matrix for the reset gate, h_{t-1} is the previous hidden state, and x_t is the current input.

2. **Update Gate (z_t):**
   - The update gate controls how much of the previous hidden state should be carried forward to the current hidden state.
   - It is calculated using the following formula:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

   - Here, σ is the sigmoid activation function, W_z is the weight matrix for the update gate, h_{t-1} is the previous hidden state, and x_t is the current input.

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

GRU step by step

3. **Candidate Hidden State ($\tilde{h}_t$):**
   - The candidate hidden state is a potential new hidden state that is calculated using the reset gate.
   - It is calculated using the following formula:

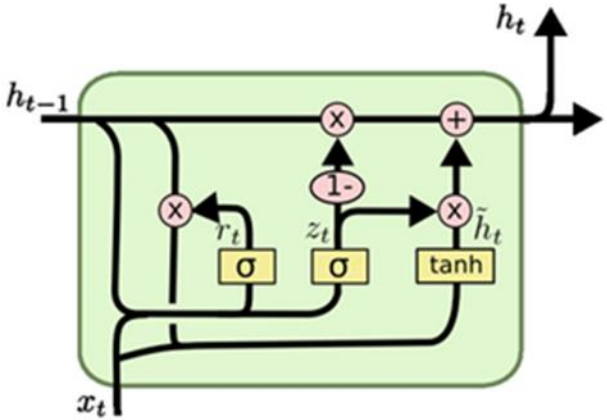
$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$

   - Here, $\tanh$ is the hyperbolic tangent activation function, W is the weight matrix for the candidate hidden state, r_t is the reset gate, h_{t-1} is the previous hidden state, and x_t is the current input.

4. **Final Hidden State (h_t):**
   - The final hidden state is a combination of the previous hidden state and the candidate hidden state, controlled by the update gate.
   - It is calculated using the following formula:

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

   - Here, z_t is the update gate, h_{t-1} is the previous hidden state, and $\tilde{h}_t$ is the candidate hidden state.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

When to use a GRU

Simplicity:

GRUs are simpler in terms of architecture compared to LSTMs. They have fewer parameters, which can lead to faster training times and less complexity in model design.

Memory Efficiency:

GRUs can be more memory efficient, with fewer gates to maintain and update than LSTMs.

Complex dependencies:

LSTMs are better suited for capturing long-term dependencies in sequences due to their more complex gating mechanism. If your problem requires capturing complex dependencies over long sequences, LSTMs may be a better choice.

Handling sequences with long-term memory:

LSTMs have been shown to perform better than GRUs in tasks that involve long-term memory retention, such as machine translation or speech recognition tasks.

When to use an LSTM

Let's have some examples with R and Python