Machine Learning

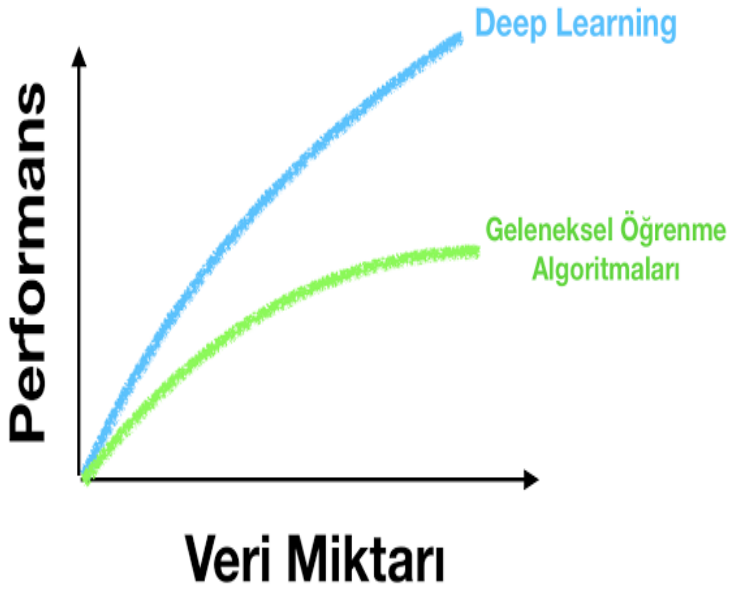
13. week

- Deep Learning
  - Convolutional Neural Network
  - Recurrent Neural Network

Why Deep Learning is so Popular ?

1. **Increase in the amount of data**

Thanks to the Internet, huge amount of data have been produced and stored digitally. Deep Learning systems gain advantage by using this big data.



Why Deep Learning is so Popular ?

2. Increase in GPU and processing power

Graphics processors are specialized equipment for parallel computing. In this way, the CPU can do some operations that are slow, much faster. Deep Learning researchers are benefitting from this increase in processing power and cheapness.

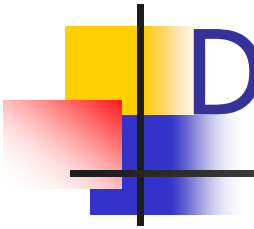
Why Deep Learning is so Popular ?

3. Depth increase

As a result of increase in processing power, deeper models are being used in practice. Deep learning models are multi layered structures.Let's look at vision system in human brain,

- The signals from the eyes through the nerves are evaluated in a hierarchical structure with several layers.
- More basic features such as edges, corners are recognized in the center where the signal is first visited after the eye.
- In later layers these edges and corners can be brought together to recognize features such as nose, mouth shapes, faces in subsequent layers and in the later layers, the appearance of the person and objects in the scene.

deep learning systems work on this principle.



Deep Learning Aprroaches

In this lesson, we will focus on two approaches:

1. Convolutional Deep Neural Networks

2. Recurrent Deep Neural Networks

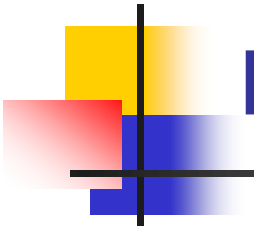
Convolutional Neural Network

- **Convolutional neural network** (**CNN**) is a class of deep, feed-forward artificial neural network.
- CNNs use a variation of multilayer perceptrons designed to require minimal preprocessing.

Convolutional Neural Network

- They have applications in
  - image and video recognition,
  - recommender systems,
  - natural language processing.

Why CNN's ? Over Ordinary neural networks ?



- Let's say we are training a classifier to identify a cat using an ordinary neural net (where we have input, hidden and output layers)

Why CNN's ? Over Ordinary neural networks ?

This is how I see

88	126	145	85	123	142	85	123	142	86	124
86	125	142	84	123	140	83	122	139	85	124
85	124	141	82	121	138	82	121	138	84	123
82	119	135	80	117	133	80	117	133	85	122
78	114	128	77	113	127	79	115	129	84	120
79	115	129	78	114	128	80	116	130	83	119
82	118	130	81	117	129	81	117	129	82	118
83	117	129	82	116	128	82	116	128	82	116
79	113	123	79	113	123	80	114	124	81	115
76	108	119	76	108	119	77	109	120	80	112
76	109	118	76	109	118	77	110	119	79	112

This is how my computer sees

Why CNN's ? Over Ordinary neural networks ?

- An ordinary neural networks typically takes features as inputs, for this problem we take image array as inputs, so we have a vector, size of (image width*height) as an input.
- We feed it to the model and train it (back propagation) for many images for many iterations.

Why CNN's ? Over Ordinary neural networks ?

- Once the network is trained then we can give another cat picture to predict (to get the score) to see if it gives the result as cat(high probability score).
- well, it works, but wait..

Why CNN's ? Over Ordinary neural networks ?

What if I gave the test pictures like these for prediction.

The ordinary network may not predict well

Why CNN's ? Over Ordinary neural networks ?

- CNNs are used mainly to look for patterns in an image, we don't need to give features, the CNN understands the right features by itself as it goes deep. This is one of the reasons why we need CNN's. Period.
- And another reason is, ordinary neural networks don't scale well for full sized images , let's say that input images size =100(width) * 100 (height) * 3 (rgb).Then we need to have 30,000 neurons which is very expensive in the network.

- Hence we need to learn CNN.

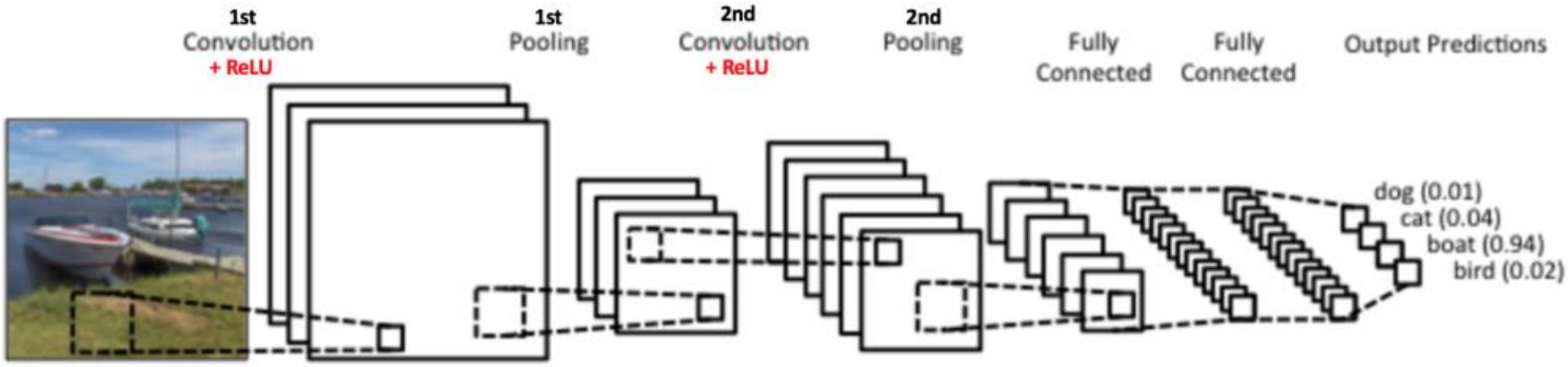
How does CNN work?

- For every image , it creates many images by applying some **filters** (just like photo editing tools)
- These **filters**, we can call **weights** , **kernels** or **features.**
- They are initialized randomly first then during the training these weights will get updated (the network learns these weights)

Design of CNN

- A CNN consists of an input and an output layer, as well as multiple hidden layers.
- The hidden layers of a CNN typically consist of
  1. Convolutional Step,
  2. Non-Linearity Step
  3. Pooling Step
  4. Fully Connected Layers

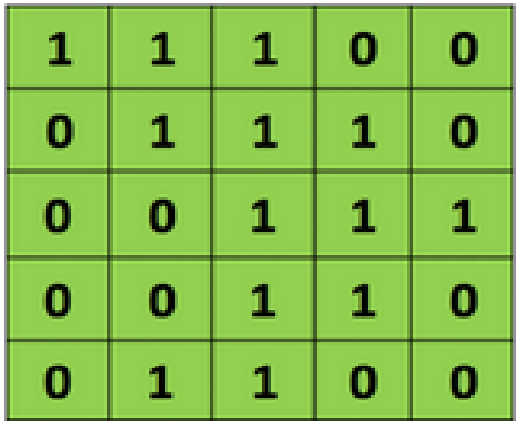
Design of CNN



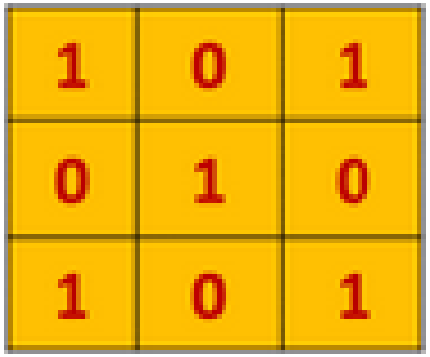
Convolution Step

- The primary purpose of Convolution in case of a CNN is to extract features from the input image.
- Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data.

Convolution Step

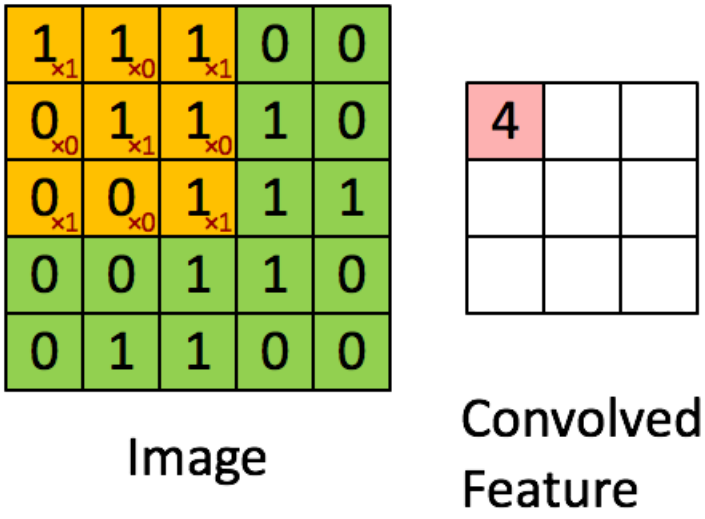


Image



Filter

Convolution Step



Convolution Step

- We slide the orange matrix over our original image (green) by 1 pixel (also called 'stride') and for every position,
- We compute element wise multiplication (between the two matrices)
- Add the multiplication outputs to get the final integer which forms a single element of the output matrix (pink).
- Note that the 3×3 matrix "sees" only a part of the input image in each stride.

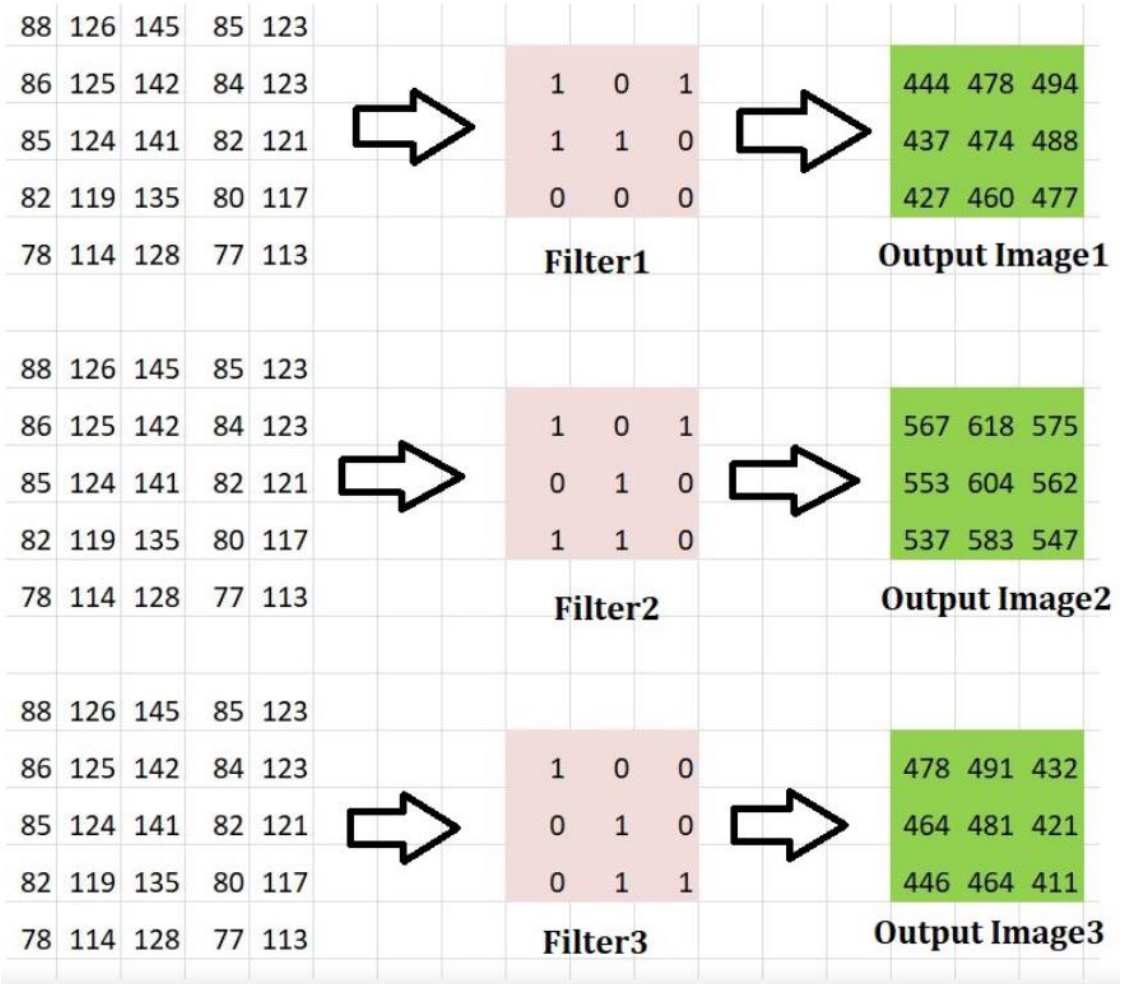
Convolution Step

- In CNN terminology, the 3×3 matrix is called a ‘filter‘ or ‘kernel’ or ‘feature detector’ and the matrix formed by sliding the filter over the image and computing the dot product is called the ‘**Convolved Feature**’ or ‘**Activation Map**’ or the ‘**Feature Map**‘.
- Note that filters acts as feature detectors from the original input image.

Convolution Step

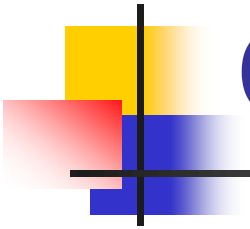
- In practice, a CNN learns the values of these filters on its own during the training process (although we still need to specify parameters such as number of filters, filter size, architecture of the network etc. before the training process).
- The more number of filters we have, the more image features get extracted and the better our network becomes at recognizing patterns in unseen images.

Convolution Step



Convolution Step

- The size of the Feature Map (Convolved Feature) is controlled by three parameters that we need to decide before the convolution step is performed:
  - Depth
  - Stride
  - Zero-padding



Convolution Step

Depth

Depth corresponds to the number of filters we use for the convolution operation.

Stride

Stride is the number of pixels by which we slide our filter matrix over the input matrix. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2, then the filters jump 2 pixels at a time as we slide them around. Having a larger stride will produce smaller feature maps.

Convolution Step

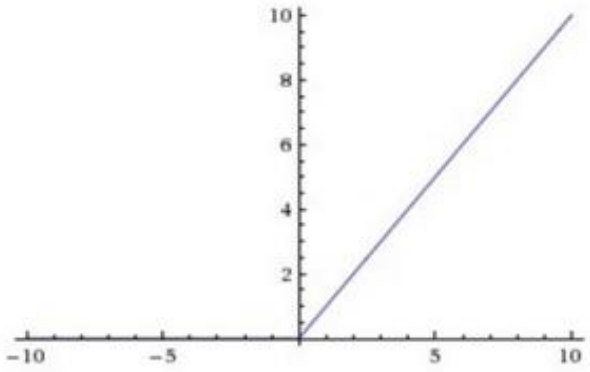
Zero-Padding

Sometimes, it is convenient to pad the input matrix with zeros around the border, so that we can apply the filter to bordering elements of our input image matrix. A nice feature of zero padding is that it allows us to control the size of the feature maps. Adding zero-padding is also called wide convolution, and not using zero-padding would be a narrow convolution.

Non-Linearity Step

- An additional operation called ReLU has been used after every Convolution operation.
- ReLU stands for Rectified Linear Unit and is a non-linear operation.

Output = Max(zero, Input)



Non-Linearity Step

- non linear functions such as
  - **tanh ,**
  - **sigmoid,**
  - **ReLU(rectified linear unit)**

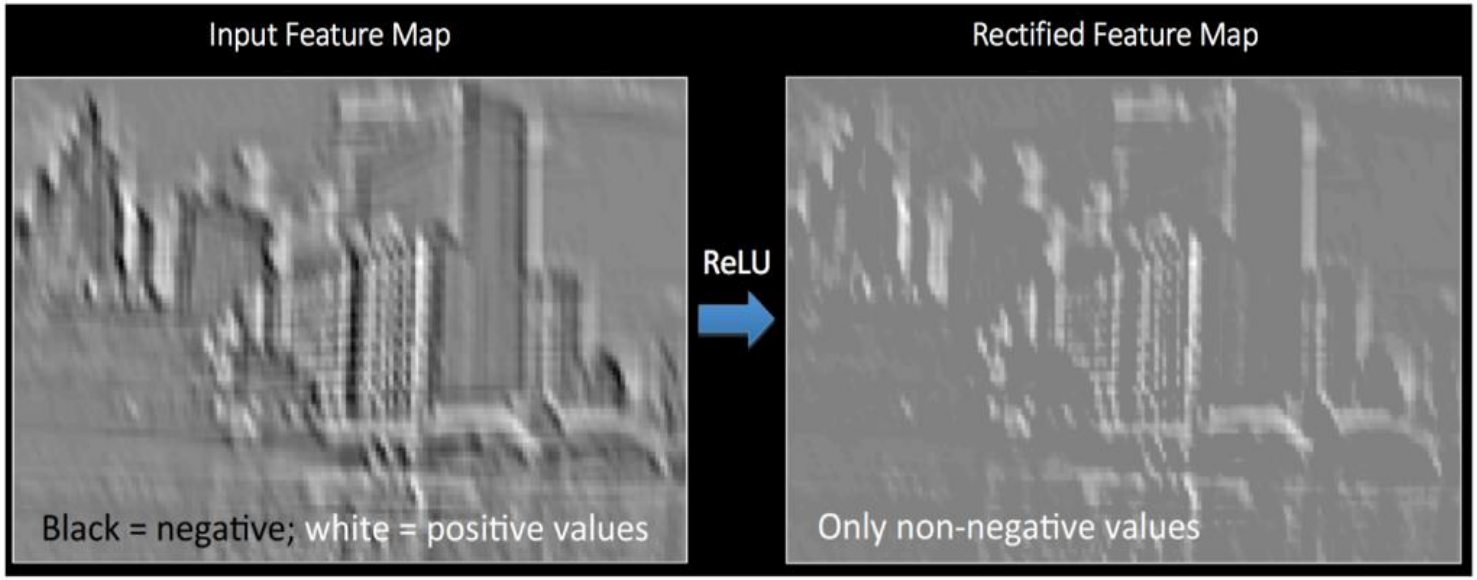
- But ReLU has been found to perform better in most situations.

Non-Linearity Step

- ReLU is an element wise operation (applied per pixel) and replaces all negative pixel values in the feature map by zero.
- The purpose of ReLU is to introduce non-linearity in our ConvNet, since most of the real-world data we would want our ConvNet to learn would be non-linear (Convolution is a linear operation – element wise matrix multiplication and addition, so we account for non-linearity by introducing a non-linear function like ReLU).

Non-Linearity Step

- It shows the ReLU operation applied to one of the feature maps obtained



The Pooling Step

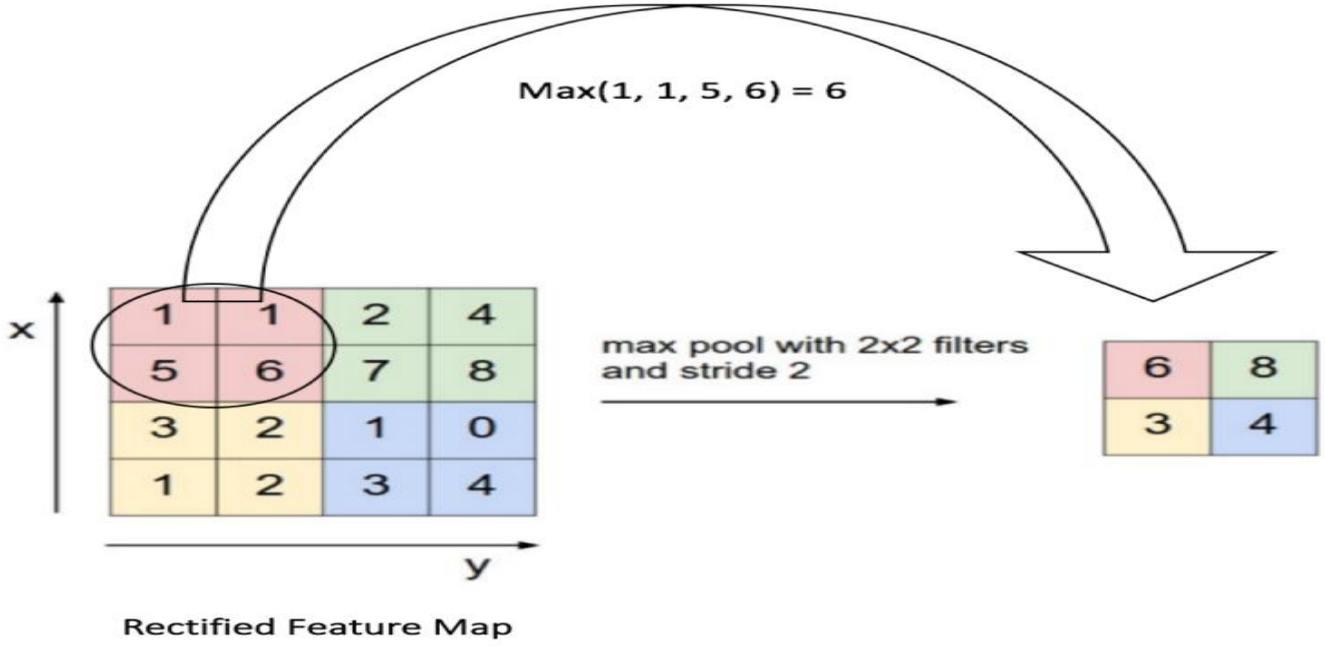
- Spatial Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map but retains the most important information.
- Spatial Pooling can be of different types: Max, Average, Sum etc.

The Pooling Step

- In case of **Max Pooling**, we define a spatial neighborhood (for example, a 2×2 window) and take the largest element from the rectified feature map within that window.
- Instead of taking the largest element we could also take the average (**Average Pooling**) or **sum** of all elements in that window.
- **In practice, Max Pooling has been shown to work better.**

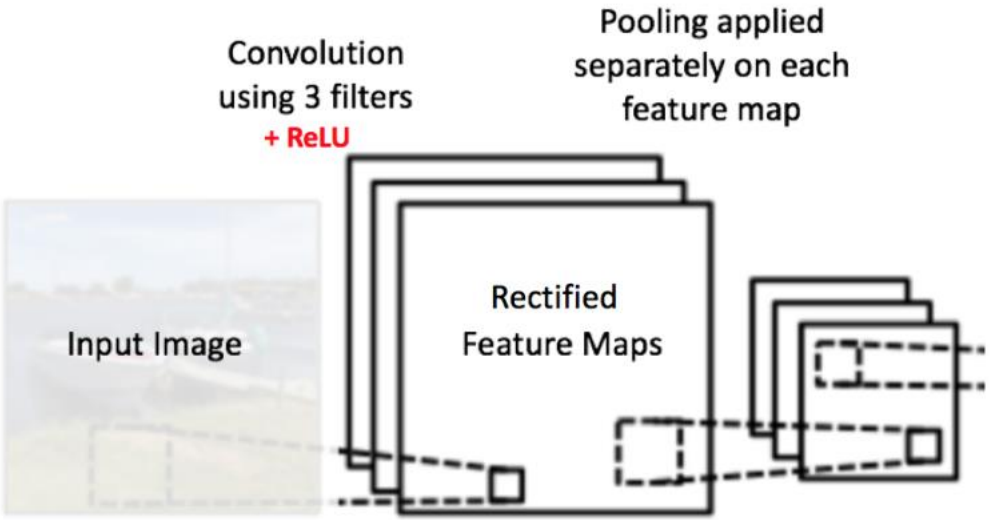
The Pooling Step

- An example of Max Pooling operation on a Rectified Feature map (obtained after convolution + ReLU operation) by using a 2×2 window.



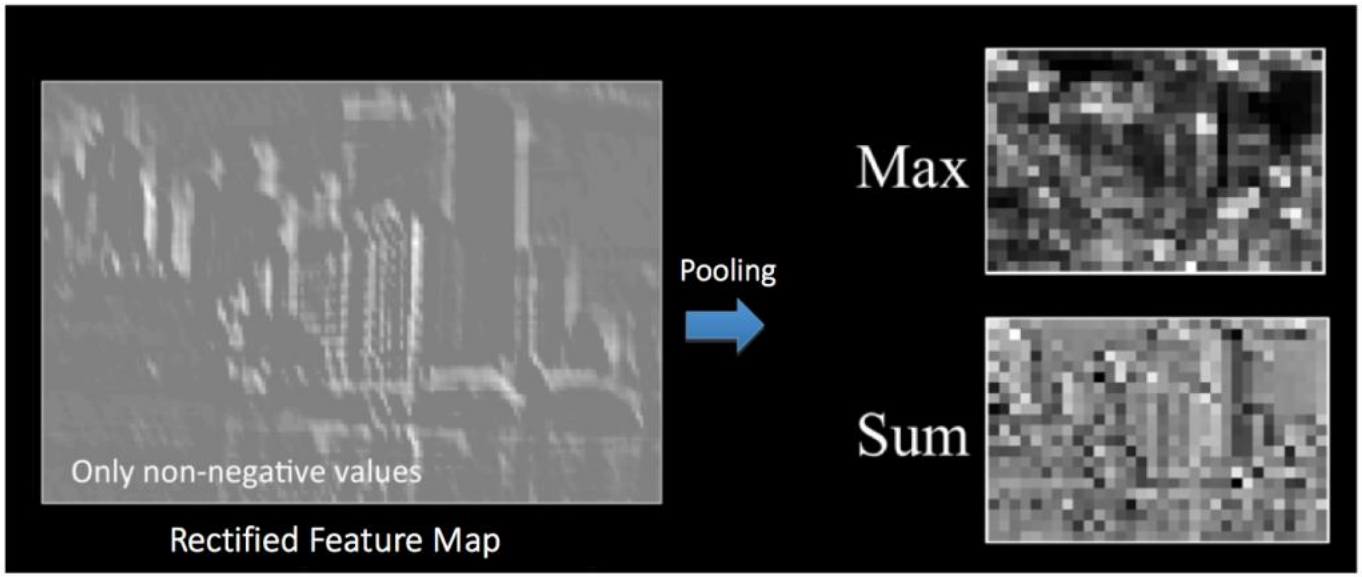
The Pooling Step

Pooling operation is applied separately to each feature map (notice that, due to this, we get three output maps from three input maps).



The Pooling Step

Max and Sum Pooling



Fully Connected Layer

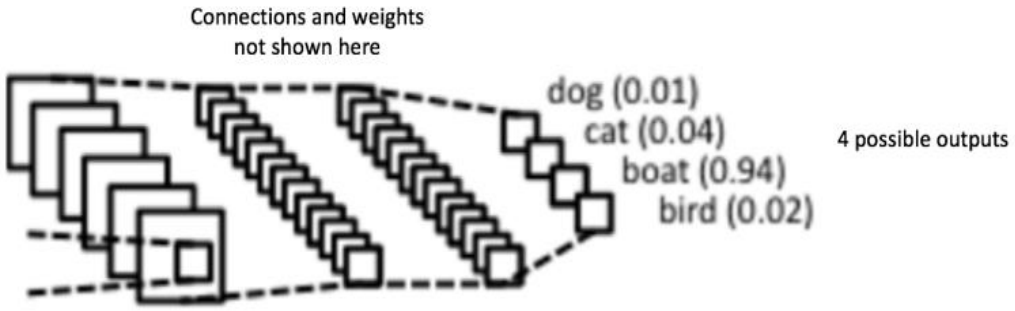
- The Fully Connected layer is a traditional Multi-Layer Perceptron that uses a softmax activation function in the output layer.
- The term "Fully Connected" implies that every neuron in the previous layer is connected to every neuron on the next layer.

Fully Connected Layer

- The output from the convolutional and pooling layers represent high-level features of the input image.
- The purpose of the Fully Connected layer is to use these features for classifying the input image into various classes based on the training dataset.

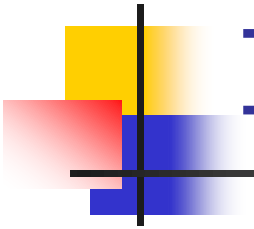
Fully Connected Layer

- The sum of output probabilities from the Fully Connected Layer is 1. This is ensured by using the Softmax as the activation function in the output layer of the Fully Connected Layer. The Softmax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one.



Fully Connected Layer

- Apart from classification, adding a fully-connected layer is also a (usually) cheap way of learning non-linear combinations of these features. Most of the features from convolutional and pooling layers may be good for the classification task, but combinations of those features might be even better.



Introduction to RNN

- Recurrent Neural Network (RNN) was introduced at first in 1986.
- Traditional feed forward network assume that all inputs and outputs are independent of each other.
- The internal state maintains a memory about history of all past inputs.

Introduction to RNN

- The network contains at least one **feed-back connection**, so the activations can flow round in a loop.
- That enables the networks to do **temporal processing** and **learn sequences.**
- Recurrent neural networks (RNNs) are often used for handling sequential data.

Example: if you want to predict the **next word** in a sentence you need to know which words came before it

Introduction to RNN

Feed forward networks

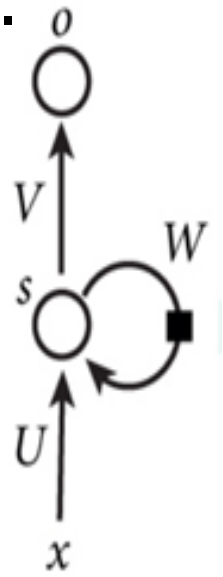
- Information only flows one way
- One input pattern produces (same) one output
- No memory of previous state

Recurrent Neural Networks

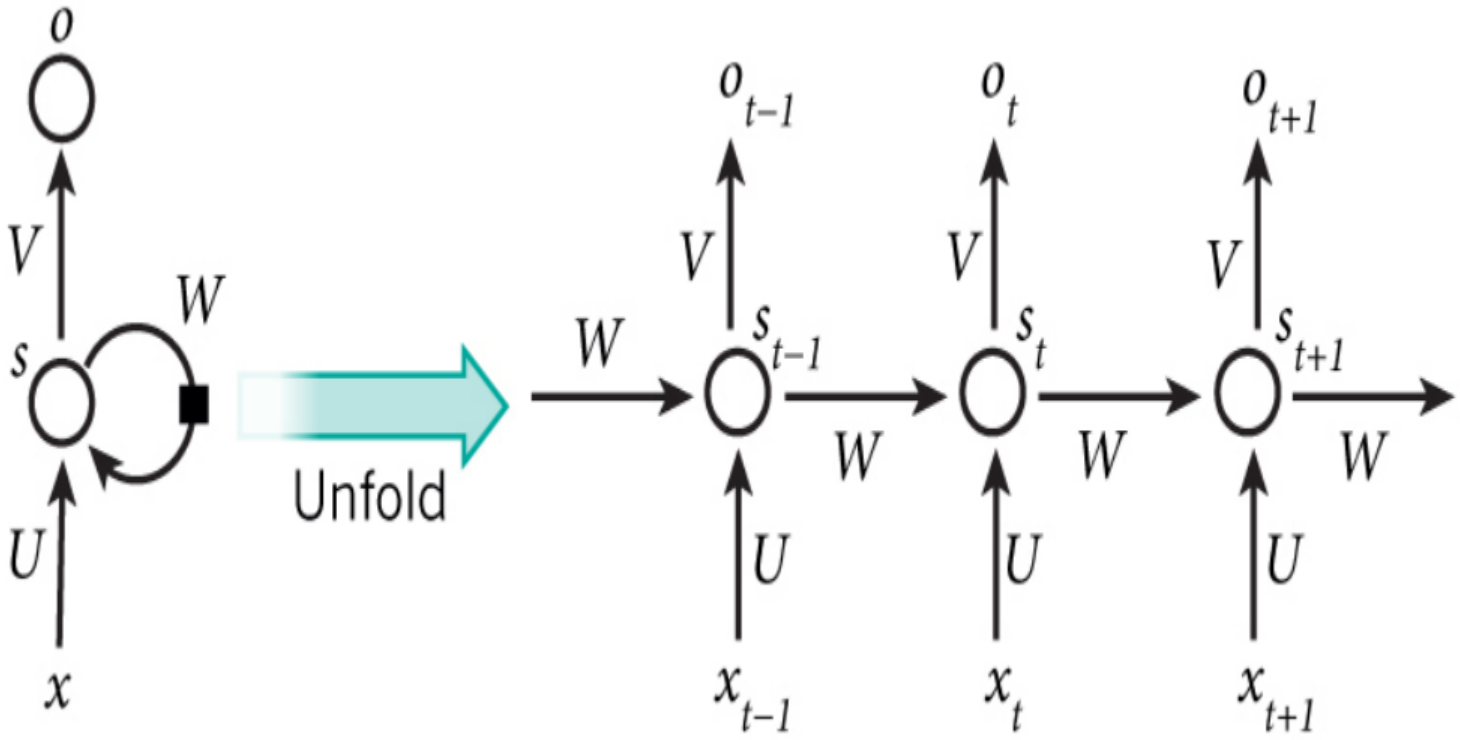
- Nodes connect back to other nodes and/or themselves
- Information flow is multidirectional
- Memory of previous state(s)
- RNNs are more "biologically realistic" because of the recurrent connectivity found in the visual cortex of the brain

RNN Architecture

- In the diagram, a chunk of neural network, s, looks at some input x_t and outputs a value o_t. A loop allows information to be passed from one step of the network to the next.



RNN Architecture



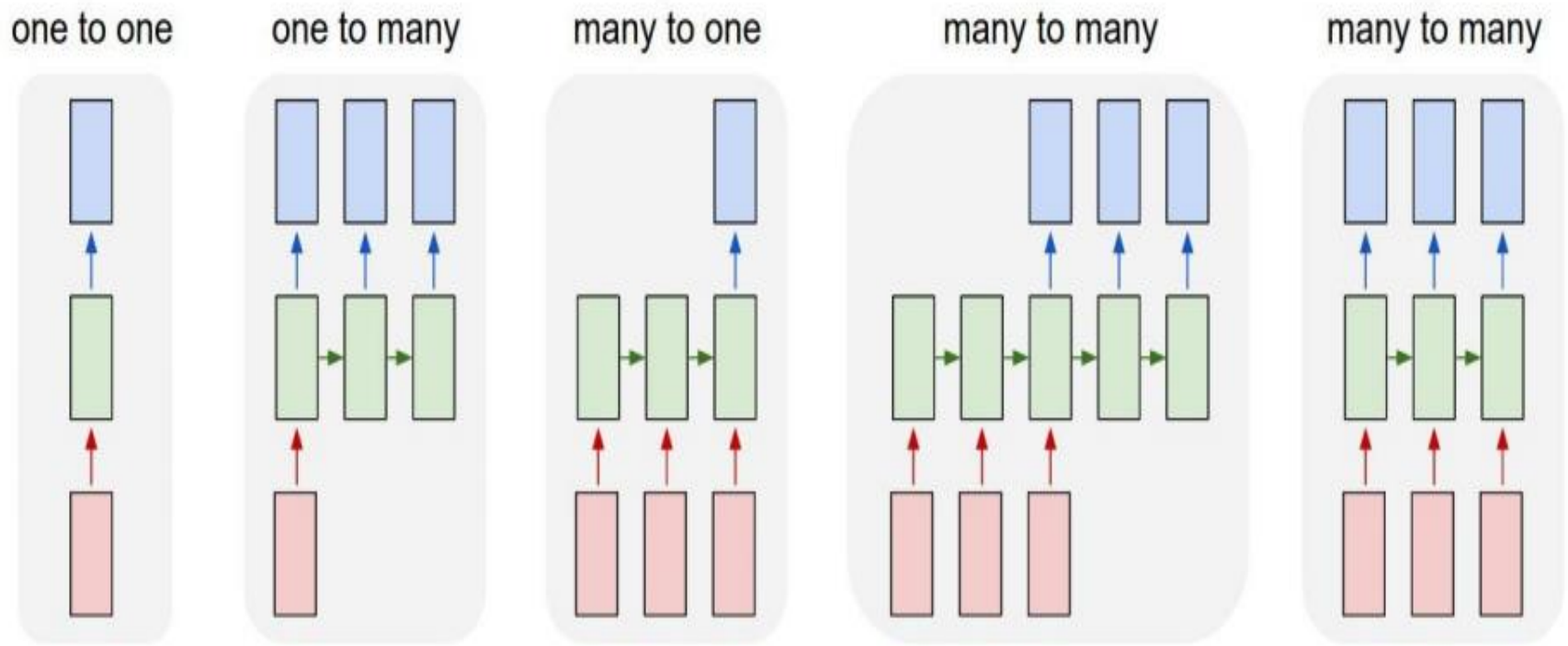
RNN Architecture

- x_t = input at time step t.
- s_t = hidden state at time step t. Calculated based on the previous hidden state and the input at the current step:

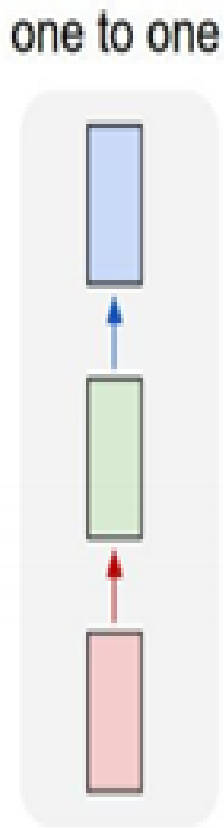
  s_t = f(U x_t + W s_{t-1}).
- f = activation function (tanh, ReLU, etc).
- s_{-1}, required to calculate first hidden state, typically initialized to all zeroes.
- o_t = output at step t.

  $o_t = W_{hy} * s_t$

RNN Architecture

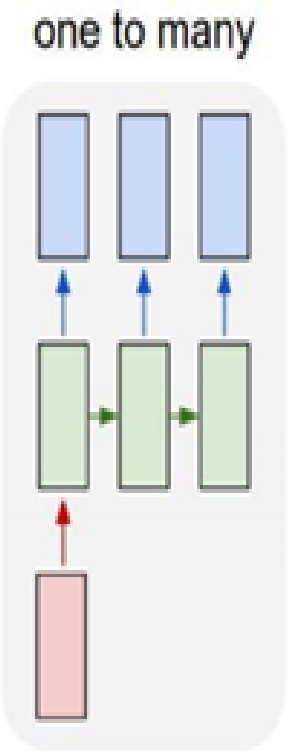


RNN Architecture



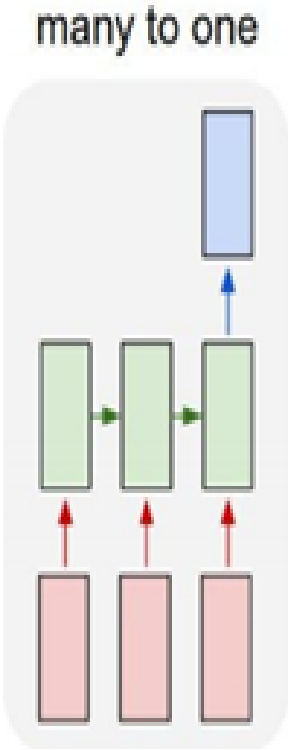
- This is in fact a type of recurrent neural network—a **one to one** recurrent net, because it maps one input to one output. A one to one recurrent net is equivalent to an artificial neural net.

RNN Architecture



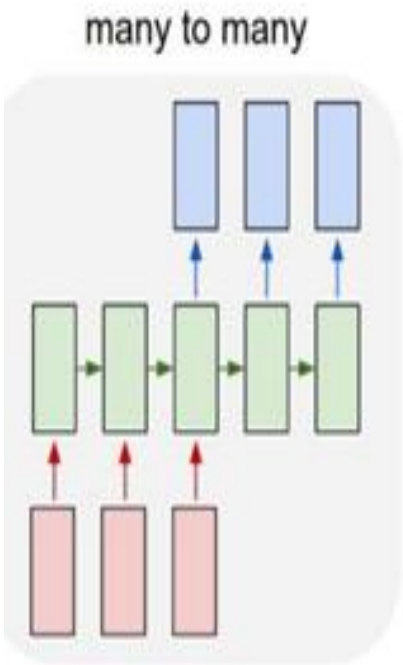
- One input is mapped to multiple outputs. An example of this would be image captioning—the input would be the image in some processed form and the output would be a sequence of words.

RNN Architecture



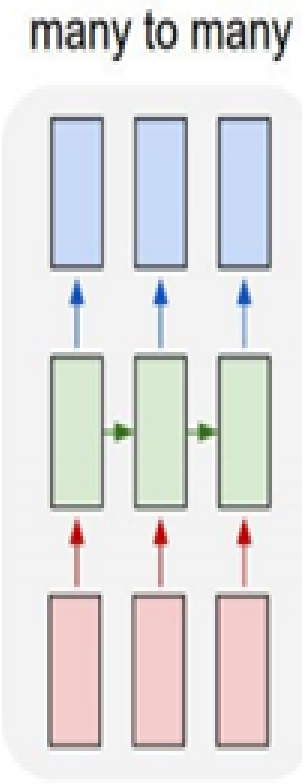
- The input is in the form of a sequence, and so the hidden states are functionally dependent on *both* the input at that time step and the previous hidden state.

RNN Architecture

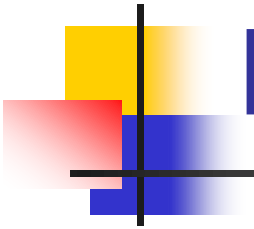


- The final type of recurrent net is many to many, where both the input and output are sequential.
- A use case would be machine translation where a sequence of **words in one language needs to be translated to a sequence of words in another.**

RNN Architecture



- Another type of many to many architecture exists where each neuron has a state at every time step, in a "synchronized" fashion. Here, each output is only dependent on the inputs that were fed in during or before it. Because of this, synchronized many to many probably wouldn't be suitable for translation.



RNN Variants

RNNs come in many variants:

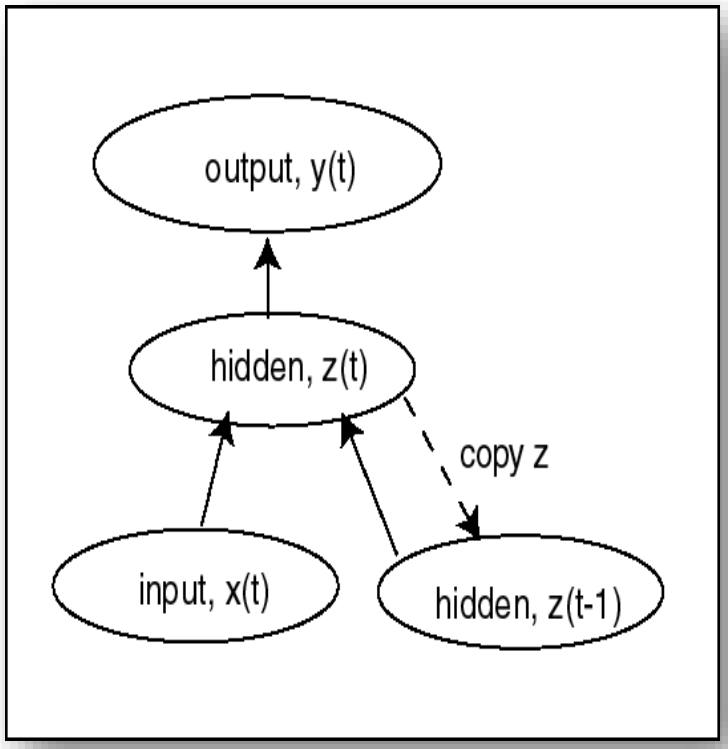
- Simple Recurrent Networks
- Long Short Term Memory (LSTM)
- Convolutional Recurrent Neural Networks

Simple RNN: Elman Network

- It first used by Jeff Elman (1990).
- The **SRN** is a specific type of back-propagation network.
- It assumes a feed-forward architecture, with units in input, hidden, and output pools.
- It also allows for a special type of hidden layer called a “context” layer.

Simple RNN: Elman Network

- Copy inputs for time t to the input units.
- Compute hidden unit activations using net input from input units and from copy layer.
- Compute output unit activations as usual.
- Copy new hidden unit activations to copy layer.



Backpropagation Through Time

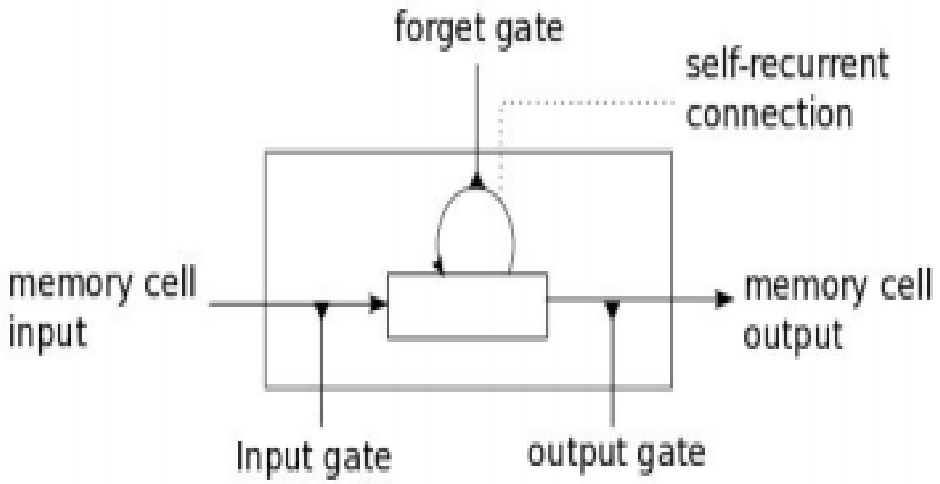
- The Backpropagation Through Time (BPTT) learning algorithm is a natural extension of standard backpropagation that performs gradient descent on a complete unfolded network.
- The more context (copy layers) we maintain, the more history explicitly include in our gradient computation.
- The downside of BPTT is that it requires a large amount of resources
  - Storage – entire history needs to be stored
  - Computation – Gradient calculations for all layers

Long Short Term Memory

- Introduced by Hochreiter & Schmidhuber (1997).
- Long-Short Term Memory (LSTM) maintain a more constant error flow in the backpropogation process.
- LSTM can learn over more than 1000 time steps , and thus can handle large sequences that are linked remotely.

Long Short Term Memory

- LSTM networks introduce a new structure called a memory cell
- Each memory cell contains three gates:
  - Input gate
  - Forget gate
  - Output gate



Long Short Term Memory

- Forget Gate

$$f = \sigma\left(x_t U^f + s_{t-1} W^f\right)$$

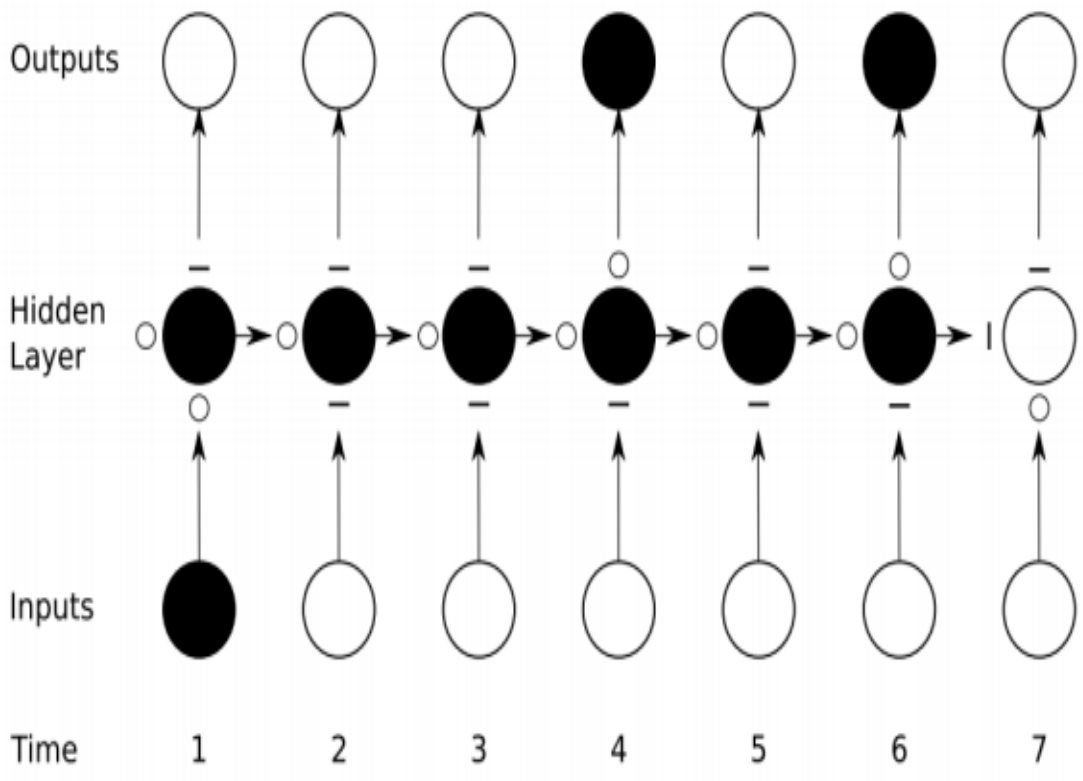
- Input Gate

$$i = \sigma\left(x_t U^i + s_{t-1} W^i\right)$$

- Output Gate

$$o = \sigma\left(x_t U^o + s_{t-1} W^o\right)$$

Long Short Term Memory



Convolutional RNN

- Convolutional neural networks (CNN) are able to extract higher level features that are invariant to local spectral and temporal variations.
- Recurrent neural networks (RNNs) are powerful in learning the longer term temporal context.
- It can be described as a modified CNN by replacing the last convolutional layers with a RNN.

Convolutional RNN

- The key module of this RCNN are the recurrent convolution layers (RCL).
- The network can evolve over time though the input is static and each unit is influenced by its neighboring units.

RNN Example

- The neural network has the vocabulary: h, e, l , o.
- That is, it only knows these four characters; exactly enough to produce the word “hello”.
- We will input the first character, “h”, and from there expect the output at the following time steps to be: “e”, “l”, “l”, and “o” respectively, to form: **hello**

RNN Example

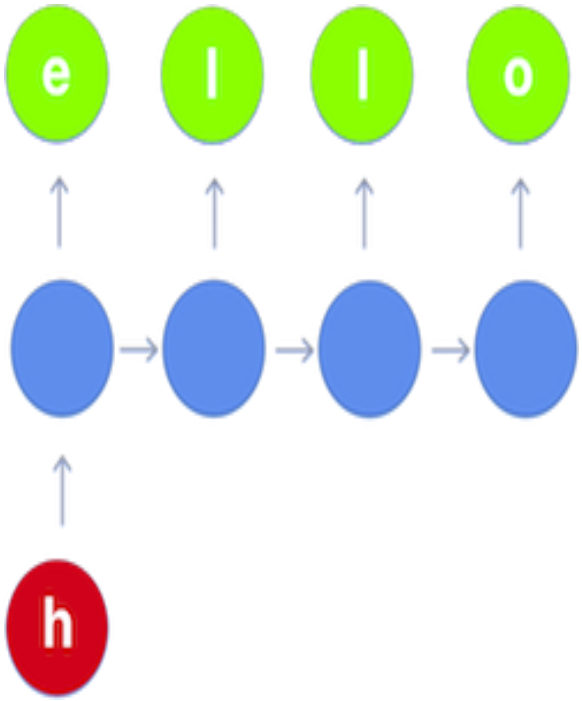
- We can represent input and output via one hot encoding, where each character is a vector with a 1 at the corresponding character position and otherwise all 0s.
- For example, since our vocabulary is [h, e, l, o], we can represent characters using a vector with four values, where a 1 in the first, second, third, and fourth position would represent “h”, “e”, “l”, and “o” respectively.

RNN Example

$$\text{"h"} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}; \text{"e"} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}; \text{"l"} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}; \text{"o"} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

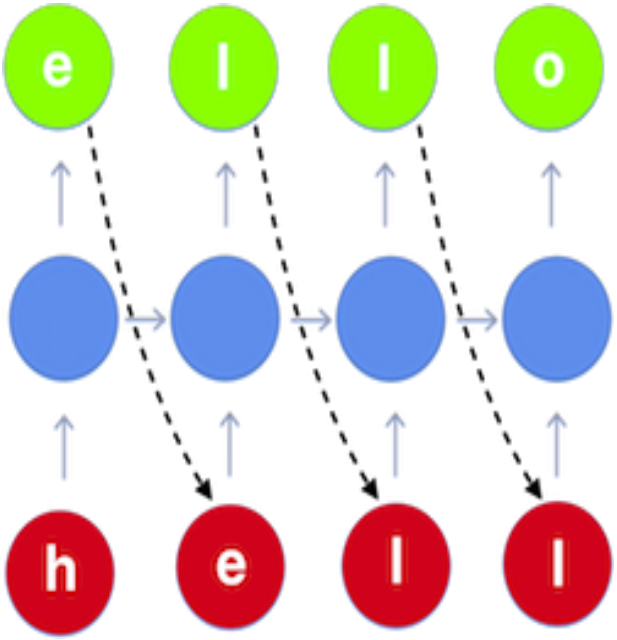
RNN Example

- As you can see, we input the first letter and the word is completed.



RNN Example

- One interesting technique would be to sample the output at each time step and feed it into the next as input:



RNN Example

- Each hidden state would contain a similar sort of vector, though not necessarily something we could interpret like we can for the output.
- The RNN is saying: given “h”, “e” is most likely to be the next character. Given “he”, “l” is the next likely character. With “hel”, “l” should be next, and with “hell”, the final character should be “o”.
- But, if the neural network wasn’t trained on the word “hello”, and thus didn’t have optimal weights (ie. just randomly initialized weights), then we’d have garble like “hleol” coming out.

CNN vs. RNN

- RNN can handle arbitrary input/output lengths.
- RNN unlike feed forward neural networks - can use their internal memory to process arbitrary sequences of inputs.
- RNNs are ideal for text and speech analysis.
- RNN will learn to recognize patterns across time.

- CNN takes a fixed size inputs and generates fixed-size outputs.
- CNN is a type of feed-forward artificial neural network.
- CNNs are ideal for images and video processing.
- CNN will learn to recognize patterns across space.