https://www.quora.com/What-is-the-advantage-of-combining-Convolutional-Neural-Network-CNN-and-Recurrent-Neural-Network-RNN

What is the advantage of combining Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN)?

There are 2 major concepts in play here:

1. CNNs capture hierarchical (and in the cases of images, spatial) information.
    1. What a CNN basically does is look at a region of the input at a time, map it to some output, and repeat this process for each region in the input.
    2. By putting a series of convolutions one after another, we get our network to learn hierarchically: each subsequent layer is a convolution of the previous layer's values.
    3. This means that the later layers in a CNN capture progressively higher level features.
    4. When you do this over an image, the network learns spatial information by combining pixels into edges, edges into shapes, to figures, and so on.
2. Recurrent Neural Networks capture temporal order, that is, order in time.
    1. One of my favourite RNN architectures is the LSTM. It is a very smart technique which selectively decides what to "remember" in a sequence and what to "forget". That is, which input will affect the state of the neuron and which won't.
    2. Using this "memory" the neuron can learn a sequence and predict subsequent values.

So now it is pretty clear what advantage you will get when you combine CNN and RNN in your model. Your model will be capable of learning spatial and temporal information in the input. This could be useful in learning embeddings for video recognition tasks where temporal information is also useful apart from spatial data.

Interesting fact: this is also the rationale given by the authors of the winning entry in the EmotiW 2016 competition. They use a hybrid network of CNN-RNN and 3D convolutional networks, combined with an audio processing pipeline to achieve the best results.

From the abstract of the paper:

In this paper, we present a video-based emotion recognition system submitted to the EmotiW 2016 Challenge… Specifically, RNN takes appearance features extracted by convolutional neural network (CNN) over individual video frames as input and encodes motion later, while C3D models appearance and motion of video simultaneously.

If you want to know how C3D "models appearance and motion of video simultaneously", or if you want to discuss anything written in the paper or in my answer, just drop a comment below.

https://stats.stackexchange.com/questions/362323/rnn-vs-convolution-1d

RNN vs Convolution 1D

Ask Question
1

Intuitively, are both RNN and 1D conv nets more or less the same? I mean the input shape for both are 3-D tensors, with the shape of RNN being (batch, timesteps, features) and the shape of 1D conv nets being (batch, steps, channels). They are both used for tasks involving sequences like time series, NLP etc. So my question here is this,

Are the steps and channels in 1D conv nets similar to the time steps and features in RNN? If they are, then why don't we use Conv 1D for time series problems instead of RNN since they are much faster compared to RNNs?

Please note that this is not a direct comparison, I know that they both work differently on an architectural level but I am just trying to get a high-level overview.

- ⇨ Yes the interpretation of the dimensions is pretty similar in both cases.

An important case where RNNs are easier to use is with data of unknown lengths. For example, in sentence translation (e.g. translating Chinese to Icelandic) both the input and output sizes are dynamic. In this case, it is easier and more intuitive to use RNNs than to try and shoehorn in CNNs.

If instead your task is closer to classification of fixed length sequences your intuition seems correct. In my experience, usually 1D CNNs are faster to train and perform better in this case.

---

https://www.quora.com/What-is-the-reason-for-applying-convolutional-neural-nets-to-time-series-instead-using-recurrent-neural-nets

What is the reason for applying convolutional neural nets to time series instead using recurrent neural nets?

Here is one example in which a CNN will be a good choice.

You have some timeseries and want to make a prediction at each datapoint. You have the entire timeseries when making inference.

An RNN will only account for past datapoints, but a CNN will look ahead as well.

CNNs are also often better at finding local patterns because the main assumption of the CNN model is that the same local patterns are relevant everywhere.

CNNs are also good for feature extraction for the same reason, making them useful for transfer learning.

If I were trying to classify bird calls, for example, I would use a CNN rather than RNN. That's because I know that a bird call has certain distinct local patterns to it and I want my model to learn those patterns efficiently, which CNNs are good at.

CNNs also tend to be more computationally efficient because there are fewer sequential calculations.

So there are plenty of cases where CNNs are a better choice and also plenty in which RNNs are a better choice (like many NLP tasks).

---

https://datascience.stackexchange.com/questions/32169/advantages-of-recurrent-neural-networks-over-basic-artificial-neural-networks

Advantages of Recurrent Neural Networks over basic Artificial Neural Networks

Ask Question

4

I have started reading Deep Learning Book, and I am having trouble understanding the advantages of RNN. This part of confuses me:

The unfodling process thus introduces two major advantages:

1. Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather then specified in terms of variable-length history of states.
2. It is possible to use same transition function f with same parameters at every time step.

These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths, rather then needing to learn a separate model for all possible time steps. Learning a single shared model allows generalization to sequence lengths that did not appear in the training set, and enables the model to be estimated with far fewer training examples than would be required without parameter sharing.

I understand the 2nd advantage. Because the computations are recurrent, the input besides the current element in the sequence is the output of the previous hidden state which has the same structure as the current hidden state, thus the shared parameters.

But I do not understand the first advantage. I cannot visualize it or mathematically prove it, or at least I dont know how. Can anyone help me with this one? And if anyone has anything else to add on the difference (advantages) of RNN over ANN I would really appreciate it. Thanks in advance!

- ⇨ okey, I will try to explain them as easy as possible.

Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather then specified in terms of variable-length history of states.

I will use a simple example for simplification but it does not lead to losing generalisation. Suppose that your task is to add to binary numbers. Numbers are stored bitwise in memory and usually for applying any arithmetic operations, they should have a same size. What you need for adding two numbers is to learn how to add zeros and ones and when to output *carray* to the next step if you use sequential learning approaches, like RNNs. In this case, it is not really important that the two numbers do not have the same shape due to the fact that they both can be resized to the biggest size by adding zeros to their most significant bits. After resizing the inputs to have the same shape you have two options, to use MLPs or RNNs. If you use MLPs what the network learns is entirely different from what an RNN learns. The former does not learn the transition of *carray* at least as the way RNN learns. Another difference is that your MLP always will be restricted to the size which it was trained while the RNN model will be able to add two numbers with even more bits. To explain it why, for MLPs, all inputs which are connected to the hidden layers or maybe output layers, have weight. Consequently, increasing the number of inputs will lead to more weights which are not trained yet. You are not allowed to input a signal which its size is not equal to the input size of the MLPs. On the contrary, RNNs are exploited in a different way.

First, you should know RNNs better. Try to think of hidden layers of anRNN. They are like usual MLPs. Their difference is that for each node, the inputs come from the previous **step's** outputs and the current time's inputs. Bear in mind that the outcomes of the previous time step are not coming from previous neurons. The reason and I guess the *main* answer to your question is that each RNN is repeated for each input in time t

. It means you have just an MLP which is used for all time step. Suppose you are at the middle of the calculations. Two inputs are 1 and 1. You RNN takes them and the carray value and outputs 1 as the result of time step t and outputs 1

to the next step as *carray*. For the next time step the same RNN is again used and takes the inputs alongside the carry which is coming from the previous step's outputs and outputs the corresponding outputs and carries.

Due to the nature of RNNs which just take the inputs of time step t

, they are capable of dealing with signals with different lengths. The reason is that The RNN is used for each time step. This behaviour is usually called unfolding the network.

shareimprove this answer
edited Sep 23 '18 at 19:33
answered Sep 23 '18 at 19:24

Media
6,80052057

- 1

  Thank you for your help! I appreciate it! – Stefan Radonjic Oct 11 '18 at 17:33

add a comment
1

The main advantage of RNN over ANN is that RNN can model sequence of data (i.e. time series) so that each sample can be assumed to be dependent on previous ones. On the contrary, ANN can not model sequence of data. So, ANN is useful if only each sample is assumed to be independent of previous and next ones (akn as iid assumption).

---

https://stackoverflow.com/questions/52020748/why-bother-with-recurrent-neural-networks-for-structured-data

Why Bother With Recurrent Neural Networks For Structured Data?

Ask Question
28

I have been developing feedforward neural networks (FNNs) and recurrent neural networks (RNNs) in Keras with structured data of the shape `[instances, time, features]`, and the performance of FNNs and RNNs has been the same (except that RNNs require more computation time).

I have also simulated tabular data (code below) where I expected a RNN to outperform a FNN because the next value in the series is dependent on the previous value in the series; however, both architectures predict correctly.

With NLP data, I have seen RNNs outperform FNNs, but not with tabular data. Generally, when would one expect a RNN to outperform a FNN with tabular data? Specifically, could someone post simulation code with tabular data demonstrating a RNN outperforming a FNN?

+125

- ⇨ In practice even in NLP you see that RNNs and CNNs are often competitive. Here's a 2017 review paper that shows this in more detail. In theory it might be the case that RNNs can handle the full complexity and sequential nature of language better but in practice the bigger obstacle is usually properly training the network and RNNs are finicky.

Another problem that might have a chance of working would be to look at a problem like the balanced parenthesis problem (either with just parentheses in the strings or parentheses along with other distractor characters). This requires processing the inputs sequentially and tracking some state and might be easier to learn with a LSTM then a FFN.

Update: Some data that looks sequential might not actually have to be treated sequentially. For example even if you provide a sequence of numbers to add since addition is commutative a FFN will do just as well as a RNN. This could also be true of many health problems where the dominating information is not of a sequential nature. Suppose every year a patient's smoking habits are measured. From a behavioral standpoint the trajectory is important but if you're predicting whether the patient will develop lung cancer the prediction will be dominated by just the number of years the patient smoked (maybe restricted to the last 10 years for the FFN).

So you want to make the toy problem more complex and to require taking into account the ordering of the data. Maybe some kind of simulated time series, where you want to predict whether there was a spike in the data, but you don't care about absolute values just about the relative nature of the spike.

---

https://stackoverflow.com/questions/20923574/whats-the-difference-between-convolutional-and-recurrent-neural-networks

What's the difference between convolutional and recurrent neural networks?

Difference between CNN and RNN are as follows:

CNN:

1. CNN takes a fixed size inputs and generates fixed-size outputs.
2. CNN is a type of feed-forward artificial neural network - are variations of multilayer perceptrons which are designed to use minimal amounts of preprocessing.
3. CNNs use connectivity pattern between its neurons and is inspired by the organization of the animal visual cortex, whose individual neurons are arranged in such a way that they respond to overlapping regions tiling the visual field.

4. CNNs are ideal for images and video processing.

RNN:

1. RNN can handle arbitrary input/output lengths.
2. RNN unlike feedforward neural networks - can use their internal memory to process arbitrary sequences of inputs.
3. Recurrent neural networks use time-series information. i.e. what I spoke last will impact what I will speak next.
4. RNNs are ideal for text and speech analysis.

---

https://wiki.tum.de/display/lfdv/Convolutional+Neural+Networks

Aller directement à la fin des métadonnées

- Créé par Simon Pöcheim, dernière modification le 31.janvier 2017

Aller au début des métadonnées

Convolutional Neural Networks

Aller directement à la fin des métadonnées

- Créé par Simon Pöcheim, dernière modification le 31.janvier 2017

Aller au début des métadonnées

Abstract

Abstract

Keywords:

Il n'existe aucune étiquette apparentée.

Motivation for Convolutional Neural Networks

Finding good internal representations of images objects and features has been the main goal since the beginning of computer vision. Therefore many tools have been invented to deal with images. Many of these are based on a mathematical operation, called convolution. Even when Neural Networks are used to process images, convolution remains the core operation.

Convolutional Neural Networks finally take the advantages of **Neural Networks (link to Neural Networks)** in general and goes even further to deal with two-dimensional data. Thus, the training parameters are elements of two-dimensional filters. As a result of applying a filter to an image a feature map is created which contains information about how well the patch corresponds to the related position in the image.

Additionally, convolution connects perceptrons locally. Because features always belong to their spatial position in the image, there is no need to fully connect each stage with each other. Convolving preserves information about the surrounding perceptrons and processes them according to their corresponding weights. In each stage, the data is additionally processed by a non-linearity and a rectification. In the end, pooling subsamples each layer.

Deep learning finally leads to multiple trainable stages, so that the internal representation is structured hierarchically. Especially for images, it turned out that such a representation is very powerful. Low-level stages are used to detected primary edges. High-level stages lastly connect information on where and how objects are positioned regarding the scene.

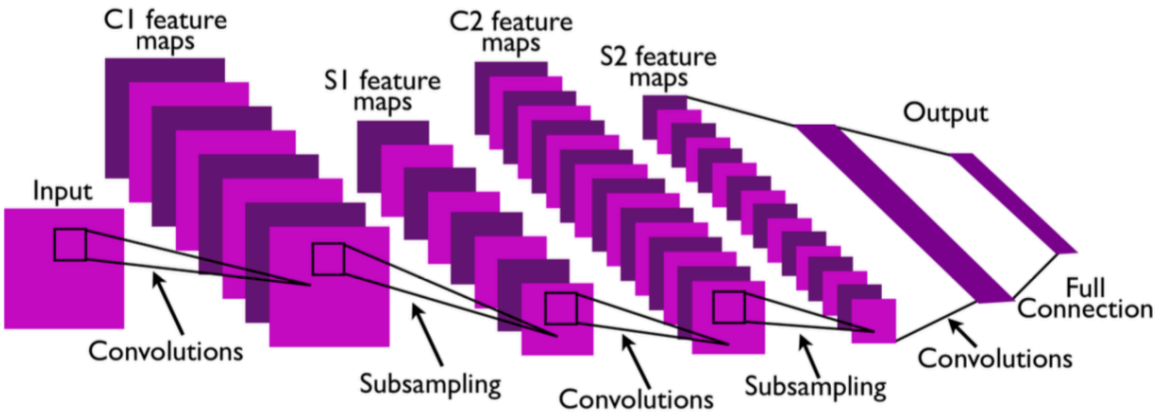


Figure 1: Typical convolutional neural network with two feature stages [2].

After introducing relevant basics in image processing and discrete convolution, the typical layers of convolutional neural networks are regarded more precisely.

Image Processing and Convolution

An image, as far as computer vision is concerned, is a two-dimensional brightness array of intensity values from 0 to 255. Thus, for a multicolor image, three intensity matrices are necessary. They are related to the color channels red, green and blue, short RGB. One channel is a map I

, defined on a compact region Ω of two-dimensional surface, taking values in the positive real numbers. So I

is a function

(1) $$I:\Omega\subset R_2\rightarrow R_+;(i,j)\mapsto I_{i,j}$$

such that one channel of the image I can be represented by a matrix of size $n_1\times n_2$

[1, p. 46].

Discrete two-dimensional convolution

Out of two functions, convolution produces a third one, by putting information of both input functions together. The result, in convolutional neural networks, called feature map and it describes how patterns of the filter are connected to the image.

Mathematically, discrete two-dimensional convolution can be described as followed. Given the filter $K\in R_{2h_1+1\,\times\,2h_2+1}$

, the discrete convolution of the image I with filter K

is given by

(2) $$(I*K)_{r,s} := \sum_{u=-h_1}^{h_1} \sum_{u=-h_2}^{h_2} K_{u,v} I_{r-u,s-v}$$

where the filter K is given by

$$K = \begin{bmatrix} K_{-h_1,-h_2} & \dots & K_{h_1,-h_2} \\ \dots & K_{0,0} & \dots \\ K_{-h_1,h_2} & \dots & K_{h_1,h_2} \end{bmatrix}.$$

Basically, in convolutional neural networks, the operation is used to match a filter with a patch of the image. This fact is represented in *figure 2*. Therefore, the resulting feature map provides information on how well the local filter fits the patch. In contrast to correlation, the filter mask is flipped horizontally and vertically. Hence, one single filter can correlate with several features at once. That's based on the associative property of convolution.

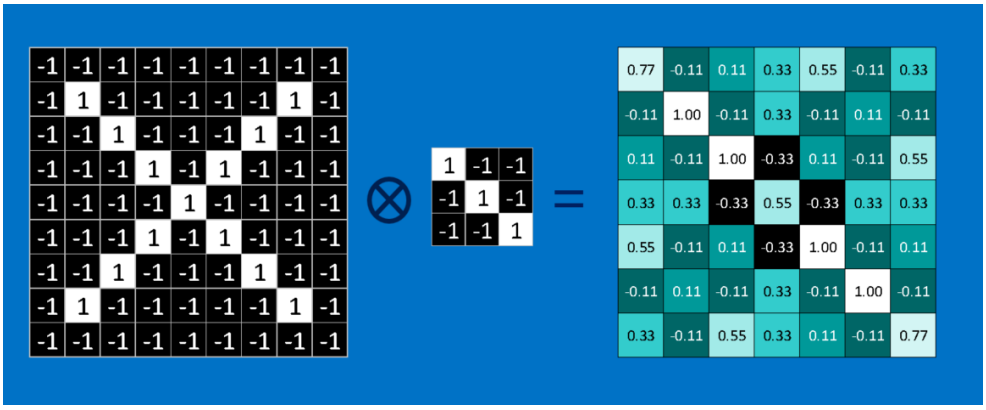


Fig. 2: Image convolved with filter (diagonal from top-left to bottom-right) to show how the resulted feature map is calculated. For reasons of simplification the image values are chosen with values of {-1; 1} [2].

Layers

A typical convolutional neural network is composed of multiple stages. Each of them takes a volume of feature maps as an input and provides a new feature map, henceforth called activation volume. The stages are consecutive separated in three layers: A convolutional layer, a ReLU layer and a pooling layer. The fully-connected layer finally maps the last activation volume onto a class of probability distributions at the output.

The following chapters will provide an overview regarding the structure and the tasks of each layer.

Convolutional Layer

The main task of the convolutional layer is to detect local conjunctions of features from the previous layer and mapping their appearance to a feature map [3]. As a result of convolution

in neuronal networks, the image is split into perceptrons, creating local receptive fields and finally compressing the perceptrons in feature maps of size $m_2 \times m_3$

. Thus, this map stores the information where the feature occurs in the image and how well it corresponds to the filter. Hence, each filter is trained spatial in regard to the position in the volume it is applied to.

In each layer, there is a bank of m_1

filters. The number of how many filters are applied in one stage is equivalent to the depth of the volume of output feature maps. Each filter detects a particular feature at every location on the input. The output $Y_{(l)i}$ of layer l consists of $m_{(l)1}$ feature maps of size $m_{(l)2} \times m_{(l)3}$. The i_{th} feature map, denoted $Y_{(l)i}$

, is computed as

(3) $$Y_{(l)i}=B_{(l)i}+\sum_{j=1}^{m_{(l-1)1}} K_{(l)i,j} * Y_{(l-1)j}$$

where $B_{(l)i}$

is a bias matrix and $K_{(l)i,j}$ is the filter of size $2h_{(l)1}+1 \times 2h_{(l)2}+1$ connecting the j_{th} feature map in layer $(l-1)$ with i_{th}

feature map in layer.

The result of staging these convolutional layers in conjunction with the following layers is that the information of the image is classified like in vision. That means that the pixels are assembled into edglets, edglets into motifs, motifs into parts, parts into objects, and objects into scenes [2] . This effect is observable in the appearance of the filters and shown in *figure 3*.

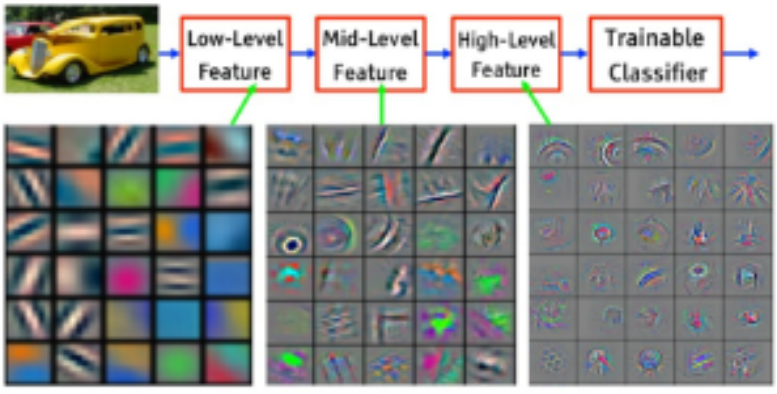


Fig. 3: Different appearance of the filters depending on the stage in the convolutional neural network.

Literature

[1] Y. Ma, S. Soatt, J. Kosecka, S.S. Sastry. An Invitation to 3-D Vision: From Images to Geometric Models. Springer New York, 2005.

[2] Y. LeCun, K. Kavukvuoglu, and C. Farabet. Convolutional networks and applications in vision. In Circuits and Systems, International Symposium on, pages 253–256, 2010.

[3] Y. LeCun, Y. Bengio, G. Hilton. Deep Learning. Nature 251, pages 436-444, May 2015.

Traditional Convolutional Neural Network Architectures

In 1990's Yann LeCun developed first application Convolutional Networks. His paper "Gradient-based learning applied to document recognition" is the documentation of first applied Convolutional Neural Network LeNet-5.

This paper is historically important for Convolutional Neural Networks.In his paper he states

"Multilayer Neural Networks trained with backpropagation algorithm consitute the best example of a successful Gradient-Based Learning technique. Given an appropriate network architecture, Gradient-Based Learning algorithms can be used to synthesize a complex decision surface that can classify high-dimensional patterns such as handwritten characters, with minimal preprocessing."

In this paper while Yann LeCun was reviewing methods for handwritten recognition, his research demonstrated that Convolutional Neural Networks outperforms other methods. This is because Convolutional Neural Networks are designed to deal with 2D shapes. [(1)] While he was researching he created LeNet, which is the first Convolutional Neural Network Architecture. In Traditional CNN Architectures we will take a look into combining modules for CNN Architectures. These combinations are based on "What is the Best Multi-Stage Architecture for Object Recognition? " another paper which was published by Yann LeCun on 2009. The next step will be taking a look into LeNet architecture.

Layers in Traditional Convolutional Neural Network Architectures

Generally, the architecture aims to build a hierarchical structure for fast feature extraction and classification. This hierarchical structure consists of several layers: filter bank layer, non-linear transformation layer, and a pooling layer. The pooling layer averages or takes the maximum value of filter responses over local neighborhoods to combine them. This process achieves invariance to small distortions.[(2)]

Traditional architecture is different from the modern ones. Here are the list and short descriptions of layers used in building models for Traditional CNNs.

- **Filter Bank Layer- F_{CSG}**

: This layer acts as a special form of convolutional layer. The only addition is that the convolutional layer is put through the tanh operation. This layer calculates the output $\mathbf{y_i}$with tanh

:

(1) $$y_j = g_i \tanh(\sum_i k_{ij} \times x_i)$$

-
- **Rectification Layer- R_{abs}**
-
- **Local Contrast Normalization Layer-N**

- **:** This layer performs local subtractive and divisive normalizations. It enforces local competition between features in feature maps and between features at the spatial location in different feature maps.
- **Average Pooling and Subsampling Layer- P_A**
-
- **Max- Pooling and Subsampling Layer- P_M**

  -

Information on Convolutional, Pooling and Rectification Layer can be found here.

Combination of Modules in Traditional Architecture:

We can build different modules by using layers. We can form a feature extraction is formed by adding a filtering layer and different combinations of rectification, normalization and pooling layer. Most of the time one or two stages of feature extraction and a classifier is enough to make an architecture for recognition.[(3)]

- F_{CSG}

-P_A: This combination is one of the most common block for building traditional convolutional networks. When we add several sequences of F_{CSG}-P_A and a linear classifier. They would add up to a complete traditional network.

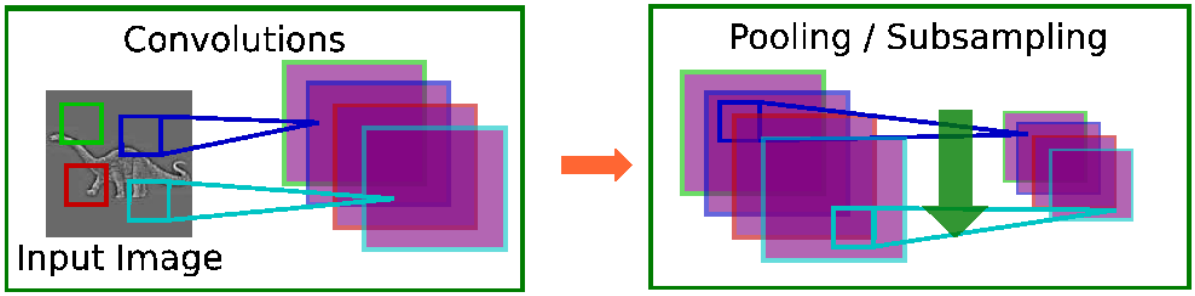


Figure 1:*the structure of* F_{CSG}-P_A

-

- F_{CSG}-Rabs-P_A: In this module the filter bank layer is followed by rectification layer and average Pooling layer. The input values are squashed by tanh, then the non-linear absolute value is calculated, and finally the average is taken and down sampled.

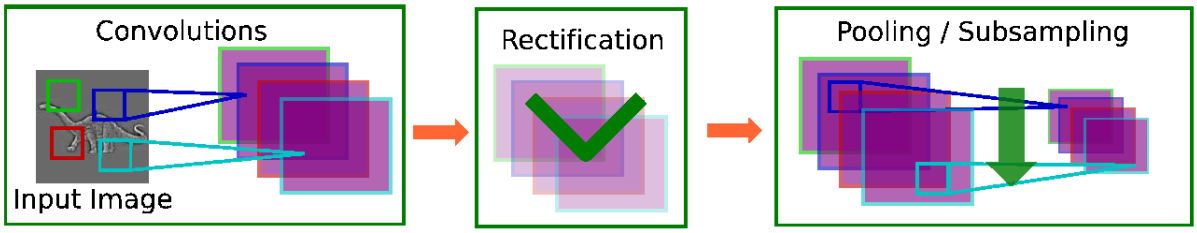


Figure 2: *the structure of* F_{CSG}-Rabs-P_A

-

- ***FCSG*-Rabs-*N*-*PA***

  - **:** This module is very similar to previous module only difference is that a local contrast normalization layer is added between rectification layer and average Pooling layer. In comparison to the previous module after the calculation of non-linear absolute value, they will be normalized and send to the pooling layer, where their average is taken and down sampled.

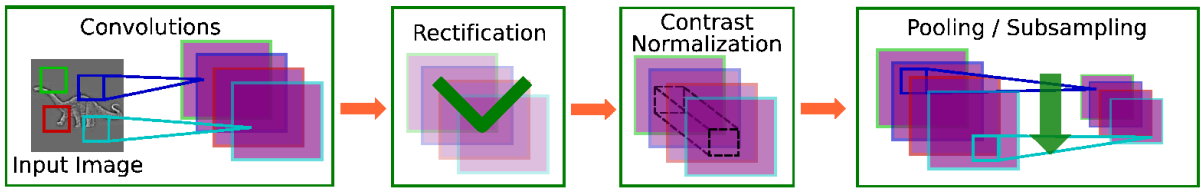


Figure 3: *the structure of* ***FCSG***

-Rabs-*N*-*PA*

(Image source[(4)]*)*

- ***FCSG***

-*PM*

- **:** This module is another common module for convolutional networks.This model forms the basis of HMAX architecture.

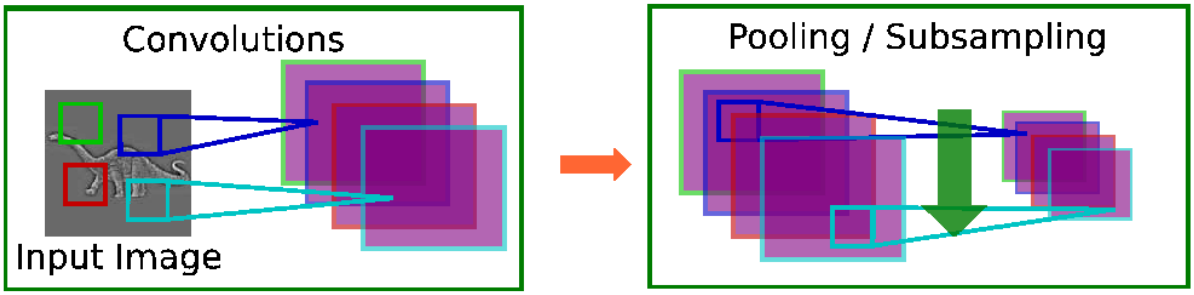


Figure 4: *the structure of* ***FCSG***

-PM

LeNet-5

LeNet-5 is the name of the first Convolutional Neural Network. It was used in Yann LeCun's experiments. LeNet-5 consists of 7 layers. These layers contain trainable weights.

The input, which is by some considered as a part of the architecture, is of a 32×32

pixel image.

The convolutional layer C1 has 156 trainable parameters, 122,304 connections and 6 feature maps. In this layer each feature map has a size of 28×28

. The main reason for this number is to prevent the number connection from input to fall below the designed boundary. Every unit in these feature maps has a connection in size of 5×5

to feature maps from the input. (or previous layer)

The sub-sampling Layer S2 has 6 feature maps. Each feature map has a size of 14×14

. Every unit in these feature maps has a connection in size of 2×2 to feature maps from the previous Layer C1. The units in S2 are send through a sigmodial function: 4 inputs coming from C1 into S2 are added, multiplied by a trainable weight, and then added to trainable bias.[7] After that operation the receptive fields with size 2×2

do not overlap and feature maps in S2 are 2 times smaller in comparison to the feature maps from C1. In total this layer has 5,880 connections and 12 trainable weights.

Layer C3 is another convolutional layer with 16 feature maps and it is similar to previous convolutional layer. Each unit in each feature map has a connection in size of 5×5

to feature maps from previous layer. These feature maps are not fully connected to previous feature maps from S2. Their connection to feature maps is show in the Figure 6. The aim of this method is to break symmetry and decrease the number of connections. In total this layer has 151,600 connections and 1,516 trainable weights.

The sub-sampling layer S4 has 6 feature maps. Each of them has a size of 5×5

. Layer S4 consists of 2,000 connections and 32 trainable weights. Each unit in the feature maps has connection in size of 2×2

to the feature maps from previous layer.

The convolutional layer C5 has 120 feature maps and 48,120 trainable connections. Every unit in these feature maps has a connection in size of 5×5

. Here every unit has a connection to all (16) feature maps from the layer S4. The feature maps of S4 are in the size of 5×5. As a result, C5 and S4 are full connection and that makes the size of the feature maps equal to 1×1. But C5 is labeled as a convolutional layer and not a fully connected layer because if C5 were to be a fully connected layer, that would make the size of feature maps be bigger than 1×1

.

Layer F6 is a full connected layer with 84 units and 10,164 trainable weights.

The main function of the output layer is to calculate Euclidean Radial Basis Function (RBF). RBF is calculated for each class, where 84 inputs are used for calculating each class. RBF unit calculates the Euclidean distance between the input vector and the parameter vector. The output functions as identifying the difference between the measurements of input pattern and our model. The bigger the difference between these vectors is bigger the RBF output.[(8)] The output is kept minimal to achieve best model. Therefore, the layer F6 is so configured that the difference would be minimized.That make F6 output close to the parameter vector.

Each RBF unit calculates the output $\mathbf{y_i}$

:

(2) $$y_j=(\sum_j x_j-w_{ij})^2$$

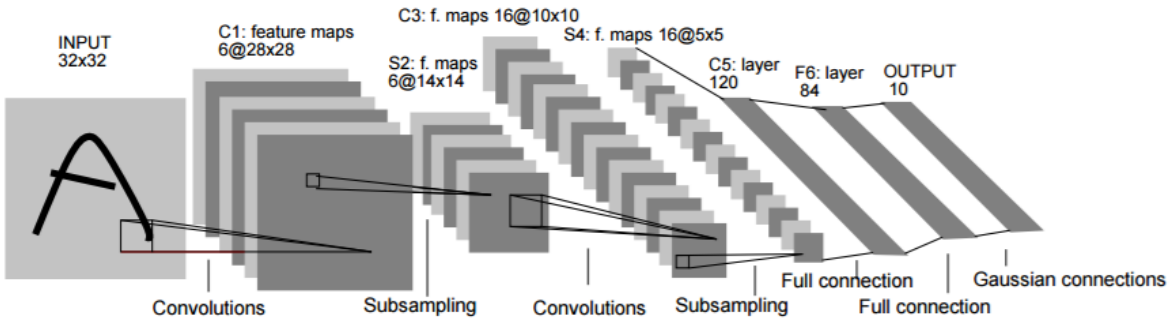


Figure 5: *the Architecture of LeNet-5 (Image source[(5)])*

	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

Figure 6: *this each column in this figure indicates which map in S2 are combined by the units in a particular map of C3 (Image source[6])*

Modern Convolutional Neural Network Architecture:

This chapter offers basic knowledge on how to build reliable simple modern architectures and demonstrates certain known examples from literature.

Layers used in Modern Convolutional Neural Networks:

Layers in modern architectures are very similar to the traditional layers, yet there are certain differences, RELU is a special implementation of Rectification Layer. You can find more information about RELU and Fully connceted Layer here.

For a simple Convolutional Network following layers are used:

- **Input Layer**
- **Convolutional Layer**
- **RELU Layer**
- **Pooling Layer**
- **Fully Connected Layer**

Main idea is that at the start the neural network architecture takes the input, which is an image size of $[A \times B \times C]$

, then at the output the class scores of the input image will be produced by this architecture. Convolutional layer and RELU (Rectification) Layer are stacked together and then they are followed by pooling layers. This structure is commonly used and repeated until the input (image) merges spatially to a small size. After that it is sent to Fully Connected Layers. The output of the last fully connected layer, which is at the end of of the architecture, produces the class scores of input image.[9]

Few examples for building Net Architecture:

- **only a single Fully Connected Layer:** This is just a linear classifier
- **Convolutional → RELU→ Fully Connected**
- **Convolutional → RELU→Pooling→ Fully Connected→ Convolutional → RELU→ Pooling→ Fully Connected→ RELU→ Fully Connected:** Convolutional Layer between every Pooling Layer
- **Convolutional → RELU→ Convolutional → RELU→ Pooling → Convolutional → RELU→ Convolutional → RELU→ Pooling→ Convolutional → RELU→ Convolutional → RELU→ Pooling→ Fully Connected→ RELU→ Fully Connected→ RELU→ Fully Connected:** This architectural form has 2 convolutional layers before each Pooling and this form is useful when building a large and deep networks because multiple convolutional layers leads to more detailed and complex features of the input before it is sent to the pooling layer, where some portion of the information will be lost.

How to build the layers:

Convolutional Layer: Generally, we want to use small filters. When building layers stacks of smaller convolutional filters are preferred over a single large layer. Assume that we have three connected 3×3

convolutional layers. In that formation neurons of the first layer have a view of 3×3 of the input, in the next layer neurons have a 3×3 view of the first layer. That means they have a 5×5view of input, the next layer neurons have a 3×3view of the second layer and a 7×7 view of input. Parameter wise this structure has $3\times(C\times(3\times3\times C))=27C2$ parameters compared to $C\times(7\times7\times C)=49C2$, which would be the case if a single 7×7

convolutional layer is used.

Pooling Layer: Max-pooling with 2×2

receptive fields eliminates 75% of the input information, because they are down sampled by 2 in height and weight. Rarely, 3×3 receptive fields are used but in general receptive fields bigger than 3×3

are not practical because that causes high loss of input data.

Specific Architectures:

AlexNet

AlexNet made Convolutional Networks popular in Computer Vision. AlexNet was developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton and won ImageNet ILSVRC challenge in 2012. During this competition it produced the best results, top-1 and top-5 error rates of 37.5% and 17.0%. [(10)]

In object recognition machine learning methods play a critical role. Generally, the aim is to improve performance of larger data sets, so more powerful models, and better techniques are used . Purpose of AlexNet is to learn from thousands of objects from millions of images. In image recognition it is common that a model does not have the whole data set, and the model should be designed so that it will have prior knowledge, in order to compensate for the missing data in model. This is the main reason AlexNet was designed using Convolutional Neural Networks.

AlexNet Architecture

As it can be seen in Figure 7, AlexNet consists of eight layers: first five of the layers are convolutional and the rest are fully connected layers. First and second convolutional layers are followed by Response-normalization layers, then these Response-normalization layers are followed by Max pooling layers. In addition the fifth convolutional layer is followed by a Max pooling layer.

The output of every convolutional layer and fully connected layer is put through RELU non-linearity. The output of the last fully connected layer sent to the 1000-way softmax layer, which produces 1000 probability values for 1000 class labels, where higher value corresponds to higher probability. Under probability distribution this neural network maximizes the average across the training cases of the log-probability of the correct label.[(11)]

As you can see in Figure 7, AlexNet consists of 2 separate pieces (In other words 2 separate GPUs). Filters of convolutional layers only have connection with filters residing on same piece. The only exception is the third convolutional layer, which is connected to all filters from the second layer of the network. And every neuron residing in a full connected layer are connected to neurons from previous layer.

Layer Size for each Layer:

Input: 224×224×3

First convolutional layer: 96

filters of size 11×11×3

Second convolutional layer: 256

filters of size 5×5×48

Third convolutional layer: 384

filters of size 3×3×48

Fourth convolutional layer: 384

filters of size 3×3×48

Fifth Convolutional Layer: 256

filters of size 3×3×192

Every fully-connected Layer has 4096

neurons

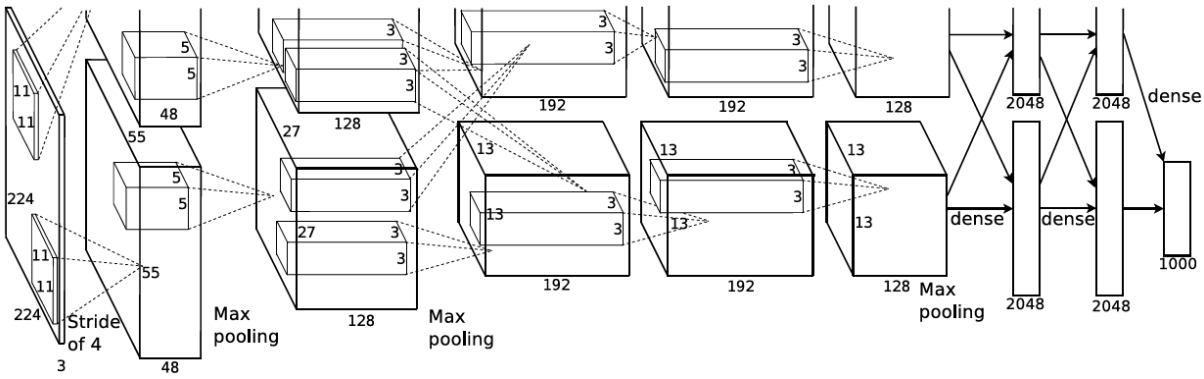


Figure 7: t*he structure of AlexNet (Image source*[(12)])

Multi-Column Deep Neural Networks Architecture

The Multi-Column Deep Neural Networks are modeled after neural layers, which reside between retina and visual cortex of mammals.This architecture offers high performance. In comparison to traditional methods, which are commonly used in computer vision and machine learning, deep artificial neural network architectures offers near-human ability and performance for recognizing handwritten digits or traffic signs. The convolutional neurons use a winner-take-all approach and yield large network depth. This large yield makes the number of sparsely connected neural layers in this deep network almost equal to the

number of neurons found between retina and visual cortex of mammals. This architectural method is the first to achieve near-human performance on MNIST handwriting benchmark and on a traffic sign recognition benchmark it outperforms humans by a factor of two.[(13)]

Multi-Column Deep Neural Networks

This architecture contains hundreds of maps per layer. This is why it is called "deep". Originally, this architectural design was inspired by Neocognition, which is an artificial neural ,proposed by Kunihiko Fukushima in 1980s. Neocognition network consisting of many layers of stacked non-linear neurons and it has been used for handwritten recognition and pattern recognition.The aim of this architecture is to iteratively minimize classification error on sets of labeled training images starting from initially random weights. Main problem with Multi-layered Deep Neural Networks was that they are hard to train and computational power to properly use such architecture was not possible. However, with recent advancements in computations and computational power that has changed.

As it can be seen in figure 8, each DNN consists of 2-dimensional fully connected layers with shared weights. They employ a winner-take-all approach the output of this layer is, then sent to pooling layer where the winning neurons are determined. The output of pooling layer is fed into 1-dimensional convolution layer.

The aim here in DNN Architecture is to train only the winner neurons, while making the other neurons not forget what their have learned. That decreases the changes per interval, and it is similar to reducing the energy consumption, if it is considered from a biological point of view. After this point, weight updates only happen after each gradient computation step, effectively making our algorithm online.

In final step, several DNN columns are combined into Multi-column DNN (MCDNN). In this Multi-column predictions from each column are averaged.Weights of each column are randomly initialized. In MCDNN the columns can be trained on the same inputs, or on inputs, which can be preprocessed indifferent ways.

Taking the average of predictions from each column:

$$y_{iMCDNN} = \sum_{j columns} y_{iDNN_j}$$

where i corresponds to the ith class and j runs over all DNN

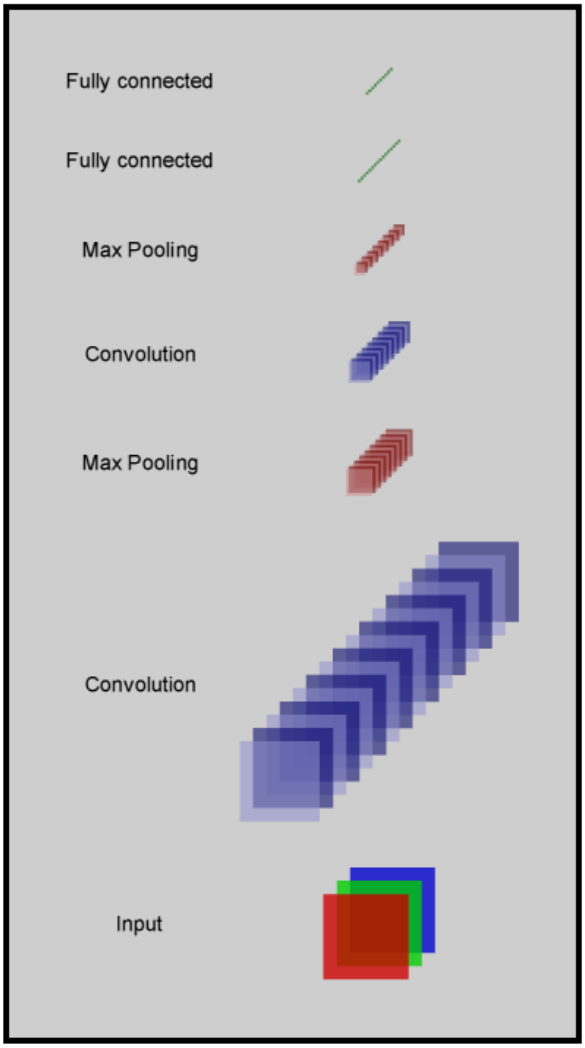


Figure 8: *the structure of each DNN (Image source[14])*

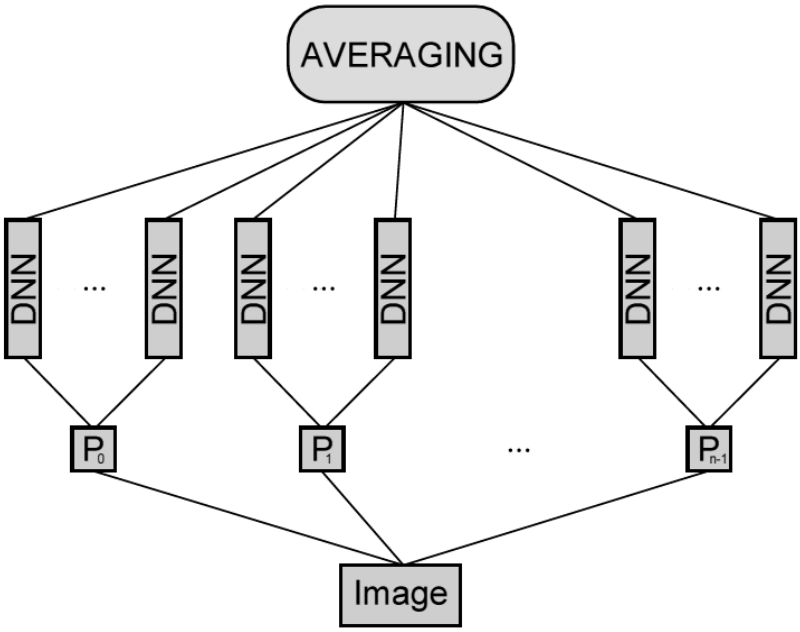


Figure 9: *the structure of each MCDNN (Image source[(15)])*

Literature

[1] Gradient-based learning applied to document recognition 1998 (Yann LeCun, P Haffner, L. Bottou, Y. Bengio)

[2]What is the Best Multi-Stage Architecture for Object Recognition? 2009 (Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato and Yann LeCun)

[3] ImageNet Classification with Deep Convolutional Neural Networks 2012 (Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton)

[4] Multi-column Deep Neural Networks for Image Classification 2012 (Dan Ciresan, Ueli Meier and Jürgen Schmidhuber)

Weblinks

[5] http://cs231n.github.io/Convolutional-networks/

https://wiki.tum.de/display/lfdv/Recurrent+Neural+Networks+-+Combination+of+RNN+and+CNN

Recurrent Neural Networks - Combination of RNN and CNN

Aller directement à la fin des métadonnées

- Créé par Lukas Wiest, dernière modification le 07.février 2017

Aller au début des métadonnées

Author: Lukas Wiest

Recurrent Neural Networks (RNN) are a class of artificial neural network which became more popular in the recent years. The RNN is a special network, which has unlike feedforward networks recurrent connections. The major benefit is that with these connections the network is able to refer to last states and can therefore process arbitrary sequences of input. RNN are a very huge topic and are here introduced very shortly. This article specializes on the combination of RNN with CNN.

Introduction

The basic idea

The basic difference between a feed forward neuron and a recurrent neuron is shown in figure 1. The feed forward neuron has only connections from his input to his output. In the example of figure 1 the neuron has two weights. The recurrent neuron instead has also a connection from his output again to his input and therefore it has in this example three weights. This third extra connection is called feed-back connection and with that the activation can flow round in a loop. When many feed forward and recurrent neurons are connected, they form a recurrent neural network (5).

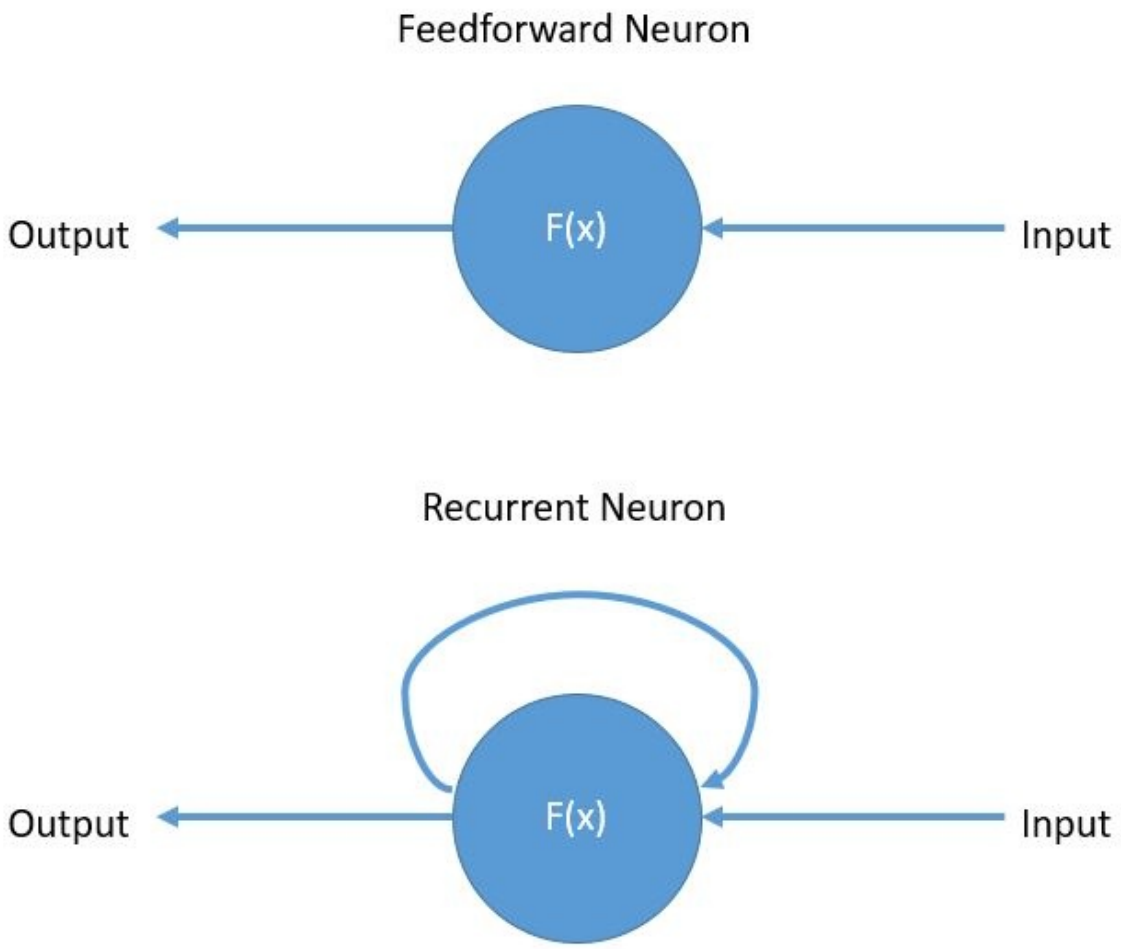


Figure 1: The basic structure of a recurrent neuron

The RNN offers two major advantages:

1. **Store Information**

   The recurrent network can use the feedback connection to store information over time in form of activations (11). This ability is significant for many applications. In (5) the recurrent networks are described that they have some form of memory.

2. **Learn Sequential Data**

The RNN can handle sequential data of arbitrary length. What this exactly means is explained in figure 2: On the left the default feed forward network is shown which can just compute one fixed size input to one fixed size output. With the recurrent approach also one to many, many to one and many to many inputs to outputs are possible. One example for one to many networks is that you label a image with a sentence. The many to one approach could handle a sequence of images (for example a video) and produce one sentence for it and finally the many to many approach can be used for language translations. Other use cases for the many to many approach could be to label each image of a video sequence (100).

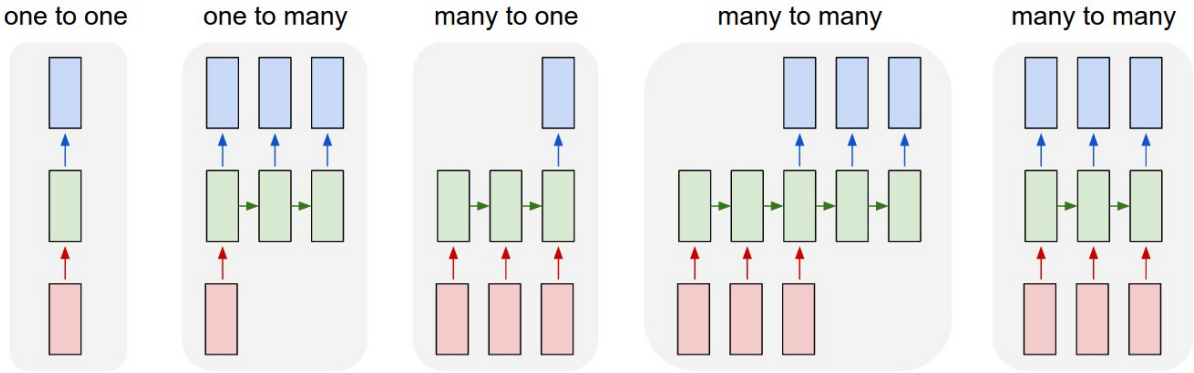


Figure 2: The different kinds of sequential data, which can be handled by a recurrent neural net (100)

Training of Recurrent Nets

The training of almost all networks is done by back-propagation, but with the recurrent connection it has to be adapted. This is simply done by unfolding the net like it is shown in figure 3. It is shown that the network consists of one recurrent layer and one feed forward layer. The network can be unfolded to k instances of f. In the example in figure 3 the network is unfolded with a depth of k = 3. After unfolding, the network can be trained in the same way as a feed forward network with Backpropagation, except that each epoch has to run through each unfolded layer. The algorithm for recurrent nets is then called Backpropagation through time (BPTT).

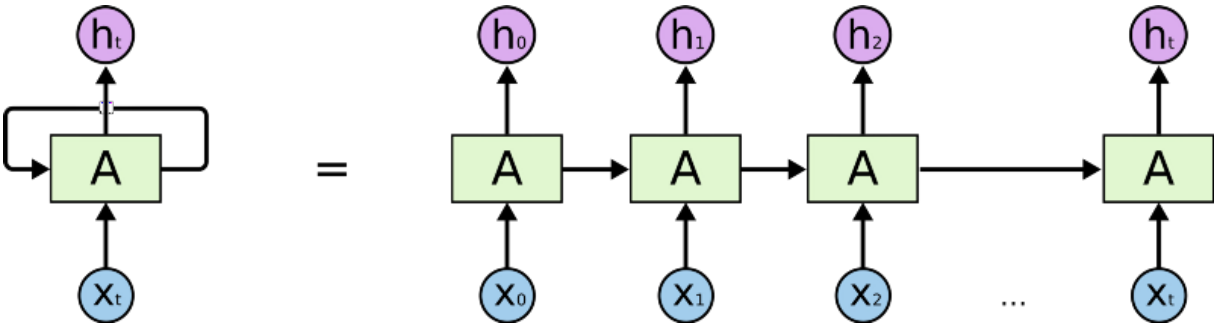


Figure 3: Unfolding of three time steps (102)

History

Recurrent Neural Networks (RNN) have a long history and were already developed during the 1980s. The Hopfield Network, which was introduced in 1982 by J.J. Hopfield, can be considered as one of the first network with recurrent connections (10). In the following years learning algorithms for fully connected neural networks were mentioned in 1989 (9) and the famous Elman network was introduced in 1990 (11). The Elman network was inspired by the architecture used by Jordan, therefore they are often mentioned together as Elman and Jordan networks. The historical architecture used by Jordan is shown in figure 4. Schmidhuber discovered in 1992 the vanishing gradient problem and therefore improved with Hochreiter the RNN to the Long Short-Term Memory (LSTM) in 1997 (8). The LSTM are more stable to the vanishing gradient problem and can better hangle long-term dependencies. Furthermore, the Bidirectional Recurrent Neural Networks (BRNN) was a further big contribution in 1997 (13). In (21) a hierarchical RNN for image processing is proposed. This biology-inspired RNN is called Neural Abstraction Pyramid (NAP) and has both vertical and lateral recurrent connections. After that no major improvement happened a long time until in 2014 the Gated Recurrent Neural Networks (GRU) were introduced, which are kind of similar to the LSTM (21). Over the last few years several people tried to combine RNN with CNN and called them sometimes RCNN. (last paragraph Combination of Recurrent and Convolutional Neural Networks)

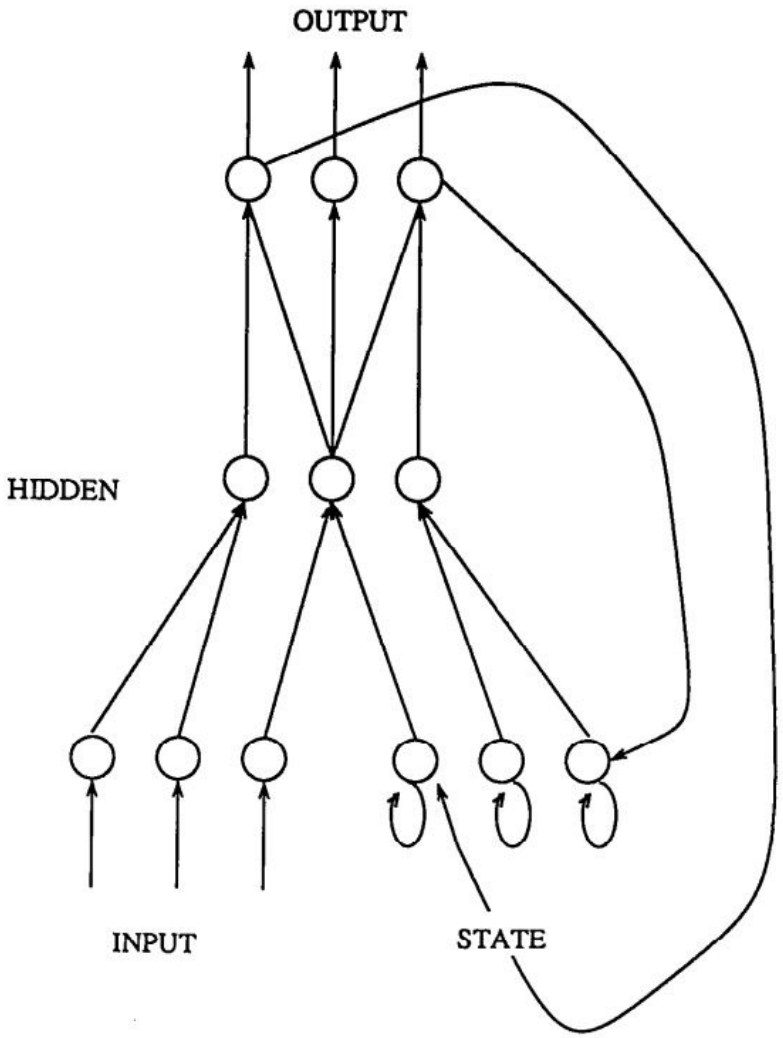


Figure 4: Historical Jordan network (1986). Connections from output to state unit are one-for-one. This network was used by Elman in his *Finding Structure in Time* paper in 1990. (11)

Why Recurrent Neural Networks?

The recurrent connections often offer advantages. They make every unit to use their context information and especially in image recognition tasks this is very helpful. As the time steps increase, the unit gets influenced by larger and larger neighborhood. With that information recurrent networks can watch large regions in the input space. In CNN this ability is limited to units in higher layers. Furthermore the recurrent connections increase the network depth while they keep the number of parameters low by weight sharing. Reducing the parameters is also a modern trend of CNN architectures:

"... going deeper with relatively small number of parameters ... " (6)

Additionally the recurrent connections yield to an ability of handling sequential data. This ability is very useful for many tasks (refer Applications). As last point recurrent connections of neurons are biological inspired and are used for many tasks in the brain. Therefore using such connections can enhance artificial networks and bring interesting behaviors. (15) The

last big advantage is that RNN offer some kind of memory, which can be used in many applications.

Applications

The ability to store information is significant for applications like speech processing, non-Markovian control and music composition (8). In addition RNN are used successfully for sequential data such as handwriting recognition and speech recognition. The big advantage in comparison to feed forward networks is, that RNN can handle sequential data as described in the paragraph before. In (2) a single RNN is proposed for sequence labeling. Most successful applications of RNN refer to tasks like handwriting recognition and speech recognition (6). They are also used in (16) for Clinical decision support systems. They used a network based on the Jordan/Elman neural network. Furthermore in (17) a recurrent fuzzy neural network for control of dynamic systems is proposed. Newer application which use combinations of RNN with CNN are for scene labeling and object detection (last paragraph).

Improvements

Long Short Term Memory (LSTM)

How does LSTM improve a Recurrent Neuron?

One major drawback of RNNs is that the range of contextual information is limited and the Back-Propagation through time does not work properly. This is noticeable in either vanishing or exploding outputs of the network. In literature this problem is called ***vanishing gradient problem*** or ***exploding gradient***. When the network is learning to bridge long time lags, it takes a huge amount of time or does not work at all, because of the vanishing gradient problem. The exploding gradient leads to oscillating weights, which also reduces the quality of the network. In practice this means when a recurrent network is learning to store information over extend time intervals, it takes a very *long* time, due to insufficient, decaying error back flow. To address exactly these problems, Hochreiter and Schmidhuber are introducing in (8) their

... novel, efficient, gradient-based method called "Long Short-Term Memory" (LSTM). (8)

The LSTM is designed to overcome the error back flow problems through carousels in their special units. This is all done with still a low computational complexity of *O(1)* and additionaly the LSTM impoves the RNN with the ability to bridge time intervals.

How does the LSTM work?

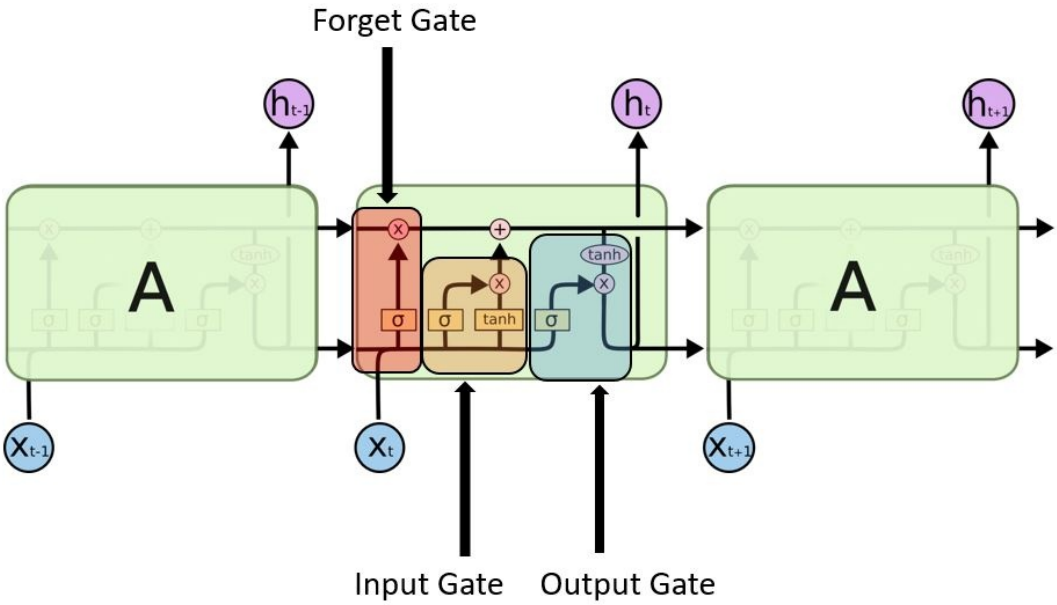
Each LSTM block consists of a forget gate, input gate and an output gate. In figure 5 on the bottom a basic LSTM cell with a step wise explanation of the gates is shown and on the top an other illustration of the cell connected into a network is shown. In the first step the forget gate looks at h_{t-1}

and x_t to compute the output f_t which is a number between 0 and 1. This is multiplied by the cell state C_{t-1} and yield the cell to either forget everything or keep the information. For example a value of 0.5 means that the cell forgets 50% of its information. In the next step the

input gate is computing the update for the cell by first multiplying the outputs i_t and $C\sim_t$ and then adding this output to the input $C_{t-1}*f_t$, which was computed in the step before. Finally the output value has to be computed, which is done by multiplying o_t with the tanh of the result of the previous step, which yields to: $h_t=o_t*tanh(C_t)$ and $o_t=\sigma*(W_o[h_{t-1},x_t]+b_o)$

. The formulas are also shown in figure 5 and are displayed in LSTM (10).

Today many people use the LSTM instead of the basic RNN and they work tremendously well on a large variety of problems. Most remarkable results are achieved with LSTM instead of RNN and many authors talk about RNN, which are using the LSTM cells.



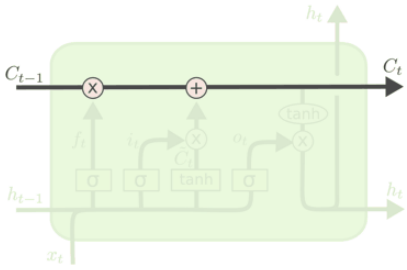


Figure 5: On the top: Different illustration of the LSTM. Sigmoid and tanh functions are shown (Based on source 102).

On the bottom: LSTM memory block. The steps of the cell. Forget, Input and Output gate with formula and active region (Based on source 102).

Connectionist Temporal Classification (CTC)

RNNs are limited in detecting cursive handwriting, where segmentation is difficult to determine. Therefore, Connectionist temporal classification is added as output layer for sequence labeling tasks. The main different step to RNN is that the network output gets transformed into a conditional probability distribution over, for example, label sequences. Then the most probable labeling for a given input sequence is chosen. The key benefits of CTC are that it does not explicitly model dependencies between labels and it obviates the need for segmented data. Furthermore, it allows to train directly for sequence labeling (18).

Gated Recurrent Unit (GRU)

The Gated Recurrent Unit was introduced in 2014 and is similar to the LSTM. It uses also the gating mechanism and is designed to adaptively reset or update its memory content. The GRU uses a reset and an update gate, which both can be compared with the forget and the input gate of the LSTM. Differently to the LSTM, the GRU fully exposes its memory at each time step and has no separate memory cells. The output balances between its last state and the new state. In (20) is shown that the LSTM and GRU outperform the traditional *tanh*-unit. However, the paper could not find a big performance difference between the LSTM and GRU (19).

Bidirectional Recurrent Neural Networks (BRNN or BLSTM)

In many cases it is useful to have access to both, the past and the future context. Therefore, Bidirectional Recurrent Neural Networks (BRNN) were introduced in 1997 by Schuster and Paliwal. In these networks the input space is increased and they have forward and backward connections. For more details, refer to (21).

Combination of Recurrent and Convolutional Neural Networks

Recurrent and Convolutional Neural Networks can be combined in different ways. In some paper Recurrent Convolutional Neural Networks are proposed. There is a little confusion abouts these networks and especially the abbreviation RCNN. This abbreviation refers in some papers to Region Based CNN (7), in others to Recursive CNN (3) and in some to Recurrent CNN (6). Furthermore not all described Recurrent CNN have the same architecture. In the following, two approaches are described in more detail. The first approach is described in the paper of Andrej Karpathy and Li Fei-Fei: They connect a CNN and RNN in series and use this for labeling a scene with a whole sentence (14). The second approach from Ming

Liang and Xiaolin Hu mixes a CNN with a RNN and use this architecture for better object detection (6).

CNN and afterwards RNN

General Structure

The alignment model described in the paper is a CNN over image region combined with a bidirectional RNN and afterwards a Multimodal RNN architecture, which uses the input of the previous net. This multimodal RNN can finally generate novel descriptions of image regions. In conclusion two single models are combined to one more powerful model, which is used to label images with sentences. To make the architecture more clear the two models are shown in figure 6. The first modal is represented as the left VGGNET and the second module is shown on the bottom right as the RNN.

How does the net work in detail?

The proposed model is trained with a set of images and their corresponding sentence descriptions. It is assumed, that the sentences written by people refer to a particular but unknown region of the image. The first model aligns sentence snippets to the visual image regions. Afterwards the second multimodal RNN gets trained with the output of the first and learn how to generate sentences. The CNN has to learn how to align visual and language data. Therefore the net uses a method described by Girshick et al. to detect objects in every image with a CNN, which is pre-trained on ImageNet. This pre-trained network is very similar to the VGGNET with the only difference, that they cut the last two fully connected layers. Karpathy and Fei-Fei propose a BRNN, that is used to represent sentences. Finally after aligning the data, the output of the first model is fed to the Multimodal Recurrent Neural Network. This Network has a typical hidden layer of 512 neurons. It is shown in figure 6 that the input of the next recurrent layer is always the output of the layer before. The network is trained to combine a word x_t

and the previous context h_{t-1} to predict a new word y_t

. For example, figure 6 shows that the first recurrent neuron outputs “straw” and therefore the next gets “straw” as input. With this input the neuron can compute the next word referring to the last. With this technique this kind of RCNN is able to create a whole meaningful sentence to describe an arbitrary image (14).

Figure 6: This image shows the architecture of a RCNN. In this case first a default CNN is used and after that a RNN is used to label the image with a sentence. (Based on source 14)

Examples of the output

In general the network generates very accurate and sensible description of images. Examples of generated images with text description are shown in figure 7. In these examples the network works pretty well except for the last two where the "wakeboard" and "two young girls" are considered as wrong. Surprisingly, the first description of the "mans in black shirt is playing guitar" does not appear in the training set. But "man in black shirt" has 20 occurrences and "is playing guitar" has 60. Therefore the network really learns how to combine these and generates a meaningful result. Although these results look very impressive, there are some limitations of the network. For example the model can only handle one specific array of pixels with fixed resolution. Furthermore this concept is based on two separate networks. Going directly from an image sentence to the region-level annotations of a single network remains an open problem (14).

Figure 7: Examples of the proposed Network (Source: 14)

Mixed CNN and RNN

Architecture

In a mixed CNN and RNN architecture the positive features of a RNN are used to improve the CNN. Liang and Hu are describing an architecture for object detection in (6) and in (2) a similar architecture for scene labeling is proposed. In these papers the combined network is called RCNN. The following quote describe what their main idea is:

> A prominent difference is that CNN is typically a feed-forward architecture while in the visual system recurrent connections are abundant. Inspired by this fact, we propose a recurrent CNN (RCNN) for object recognition by incorporating recurrent connections into each convolutional layer. (6)

The key module of this RCNN are the recurrent convolution layers (RCL), which introduce recurrent connection into a convolution layer. With these connections the network can evolve over time though the input is static and each unit is influenced by its neighboring units. This property integrates the context information of an image, which is important for object detection. The importance of context information is shown in figure 8. In this figure it is very hard to recognize the mouth or the nose without the context (6).

Figure 8: Describe the importance of context information to recognize the nose or the mouth. (6)

The network is trained with BPTT and therefore with the same unfolding algorithm described in the first paragraph (Introduction). The unfolding facilitates the learning process and sharing weights reduce the parameters. In figure 9 on the left the unfolding for three time steps of the recurrent connection is shown and on the right the architecture of the whole used RCNN. The network consists of first a convolutional layer to save computations, which is followed by a max pooling layer. On top of that two RCL, one max pooling and then again two RCL layer are used. Finally one global max pooling and a softmax layer are used. The pooling operation

have stride two and size three. The global max pooling layer outputs the maximum over every feature map, yielding to a feature vector that represents the image.

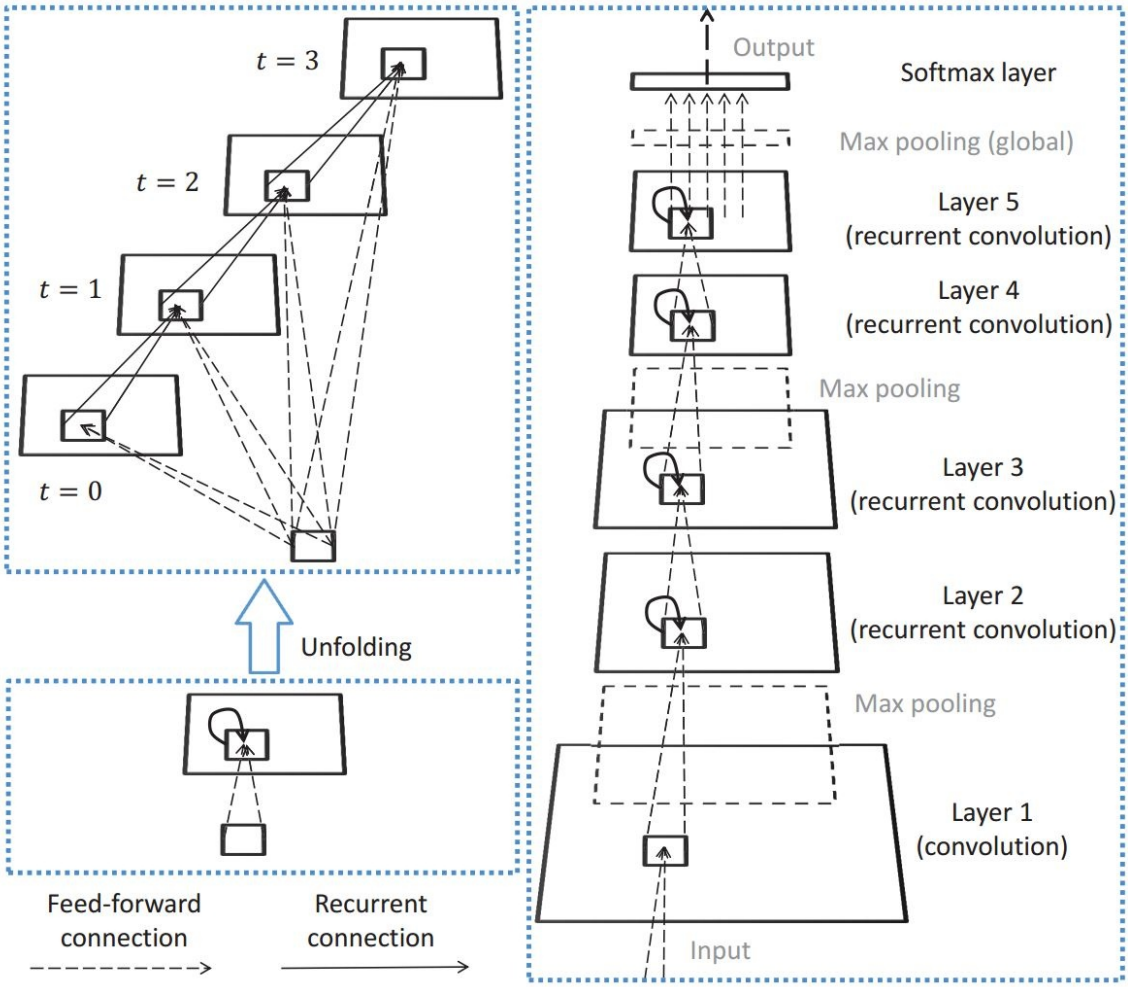


Figure 9: Mixed CNN and RNN architecture. On the left the a RCL is unfolded for three time steps, leading to a feed-forward network with largest depth of four and smallest depth of one.On the right the RCNN used in paper (6) is shown with one convolutional layer, four RCL, three max pooling and one softmax layer. (Source 6)

Results of the network

The authors show in their paper that the recurrent connections perform better as a CNN with the same number of layers. For example, if the RCL in the RCNN uses three time steps for unfolding, they added three feed forward layers to the CNN and compared these networks. This comparison indicates that the multi-path structure of the RCL is less prone to overfitting and performs better than the extended CNN. In their next experiment they compared this RCNN with state-of-the-art models, which is shown in figure 6. The layers one to five (Figure 4) are constrained to have the same number of feature maps K. Thereby RCNN-K denotes a network with K feature maps in layer one to five and RCNN-96 has for example 96 feature maps. Table 1 compares some results on the CIFAR-10 dataset. This dataset consists of 60000 color images of 32x32 pixels in ten classes. For the RCNN 50000 images were used for training and 10000 images were used for testing. The last 10000 images of the training set were used for validation of the net. The RCNN has still remarkable results compared to many

other nets. It has with a very low number of parameter very good results and the RCNN-160 reduces the testing error to 7.09%. The table shows state-of-the-art networks and compared to other networks the RCNN performs pretty good but there are some better performing nets. For example the Deep Residual Network which is also introduced in the wiki. Currently the best performing net on the CIFAR-10 dataset is the Fraction Max Pooling (4), which achives a testing error of 3.47%.

Model	Number of Parameter	Testing Error (%)
Without Data Augmentation		
Maxout	> 5 M	11.68
Prob maxout	> 5 M	11.35
NIN	0.97 M	10.41
RCNN-96	0.67 M	9.31
RCNN-128	1.19 M	8.98
RCNN-160	1.86 M	8.69
RCNN-96 (no Dropout)	0.67 M	13.56
With Data Augmentation		
Maxout	> 5 M	9.39
Prob maxout	> 5 M	9.38
NIN	0.97 M	8.81
RCNN-96	0.67 M	7.37
RCNN-128	1.19 M	7.24
RCNN-160	1.86 M	7.09
Deep Residual Networks	1.7 M	6.43
Fraction Max Pooling (4)	-	3.47

Table 1: Results of the Recurrent CNN compared to other state of the art solutions, (based on source 6 and added some relevant networks)

Conclusion

Recurrent Networks are very exciting and have already a very long history. In this history there researchers were able to get a good understanding and feeling about the recurrent network. The fact that it is biological inspired is very promising for getting better performance out of RNN. Furthermore the basic idea of the RNN evolved over the time and many remarkable contributions were made. For example the LSTM, which enhances many properties of the basic RNN. In future it can be assumed that the combination of RNN with

other networks, especially the CNN, will be continued. The improvement and the ability to handle sequential data enhance the CNN a lot and brings new unexplored behavior. This is an exciting and promising area of artificial intelligence.

---

https://towardsdatascience.com/introduction-to-sequence-models-rnn-bidirectional-rnn-lstm-gru-73927ec9df15

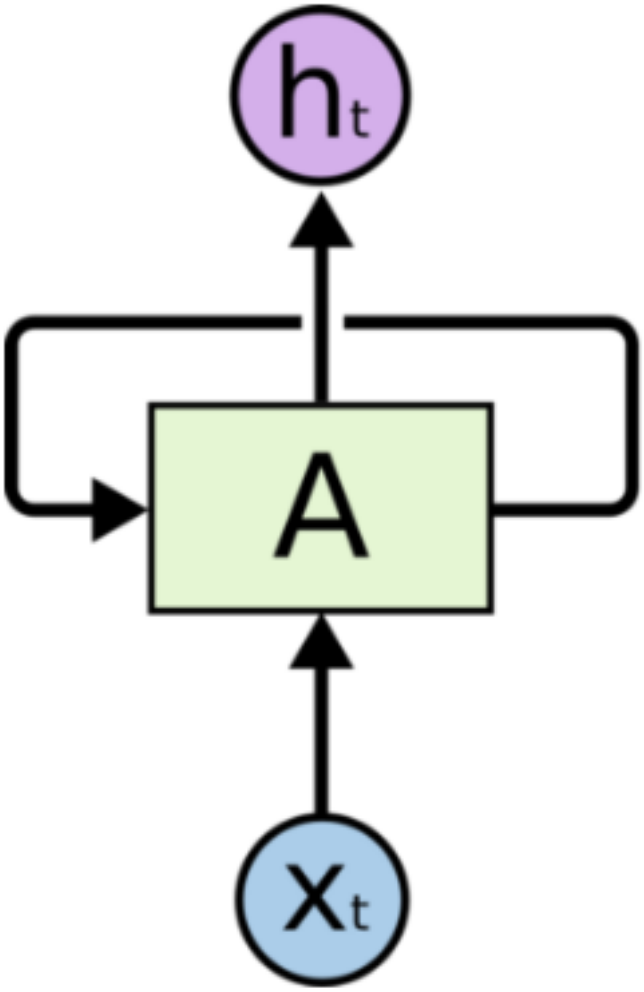
Introduction to Sequence Models—RNN, Bidirectional RNN, LSTM, GRU

A brief explanation

Introduction—Why do we need Sequence Models??

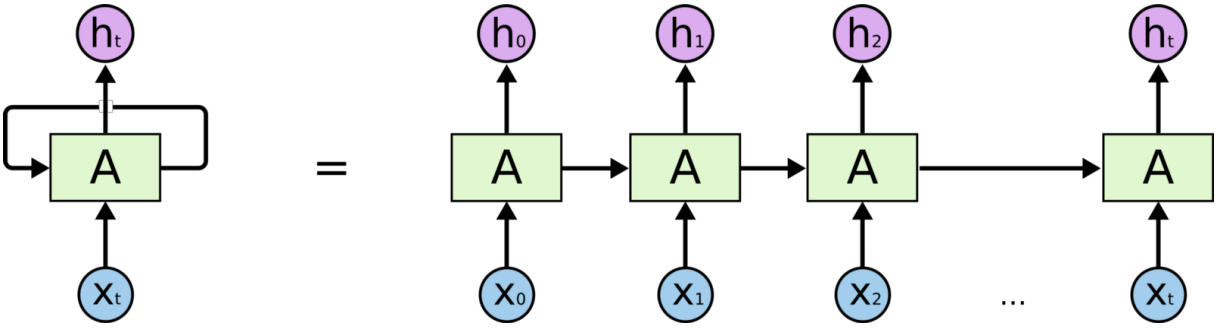
Why do we need sequence models when we already have feedforward networks and CNN? The problem with these models is that they perform poorly when given a sequence of data. An example of sequence data is an audio clip which contains a sequence of spoken words. Another example would be a sentence in English which contains a sequence of words. Feedforward networks and CNN take a fixed length as input, but, when you look at sentences, not all are of the same length. You could overcome this issue by padding all the inputs to a fixed size. However, they would still perform worse than an RNN because those conventional models do not understand the context of the given input. This is where the major difference between sequence models and feedforward models lies. Given a sentence, when looking at a word, sequence models try to derive relations from the previous words in the same sentence. This is similar to how humans think as well. When we are reading a sentence, we don't start from scratch every time we encounter a new word. We process each word based on the understanding of the previous words we have read.

Recurrent Neural Network



1. Recurrent Neural Network

The recurrent neural network is represented as shown in the above figure. Each node at a time step takes an input from the previous node and this can be represented using a feedback loop. We can unfurl this feedback loop and represent it as shown in the figure below. At each time step, we take an input x_i and a_i-1(output of the previous node) and perform computation on it and produce an output h_i. This output is taken and given to the next node. This process continues until all the time steps are evaluated.



2. Recurrent Neural Network

The equations describing how the outputs are calculated at each time step is represented below.

Let a_t represent the ouput from the previous node

$$a_t = f(h_{t-1}, x_t)$$

$$g(x) = \tanh x$$

$$a_t = g(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$$

$$a_t = \tanh W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t$$

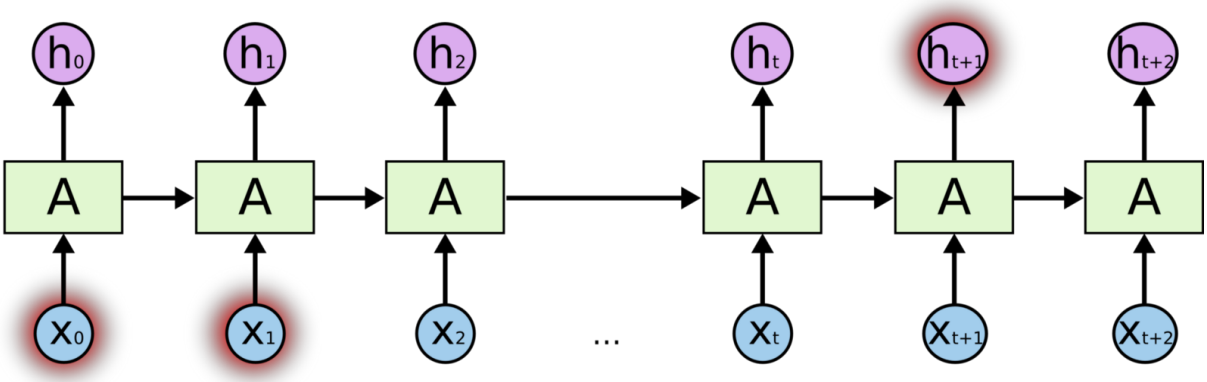
$$h_t = W_{hy} \cdot a_t$$

3. Recurrent Neural Network

Backpropagation in recurrent neural networks occurs in the opposite direction of the arrows drawn in figure 2. Like all other backpropagation techniques, we evaluate a loss function and obtain gradients to update our weight parameters. The interesting part of backpropagation in RNN is that backpropagation occurs from right to left. Since the parameters are updated from final time steps to initial time steps, this is termed as backpropagation through time.

Long Short-Term Memory—LSTM Network

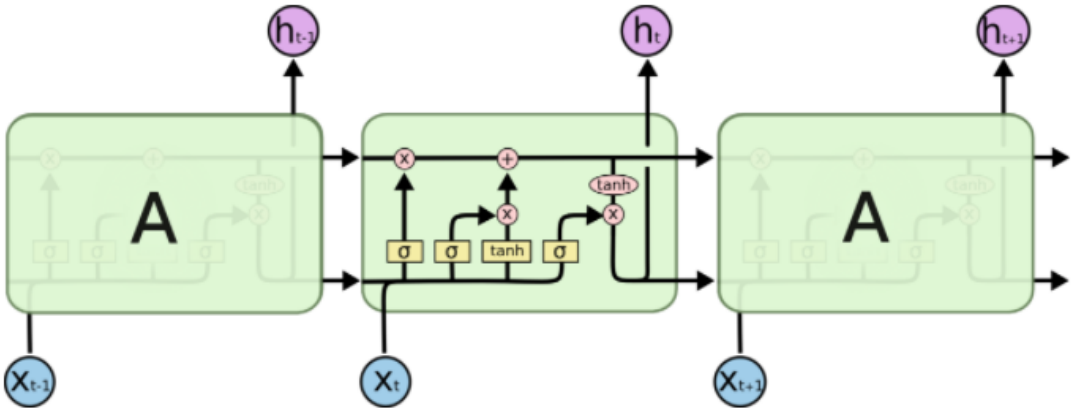
The disadvantage with RNN is that as the time steps increase, it fails to derive context from time steps which are much far behind.



4. Recurrent Neural Network

To understand the context at time step t+1, we might need to know the representations from time steps 0 and 1. But, since they are so far behind, their learned representations cannot travel far ahead to influence at time step t+1. Ex: "I grew up in France ….. I speak fluent French", to understand that you speak French, the network has to look far behind. But, it is not able to do so and this problem can be attributed to the cause of vanishing gradients. Therefore, RNN is able to remember only short-term memory sequences.

To solve this problem, a new kind of network was introduced by Hochreiter & Schmidhuber called Long Short-Term Memory.

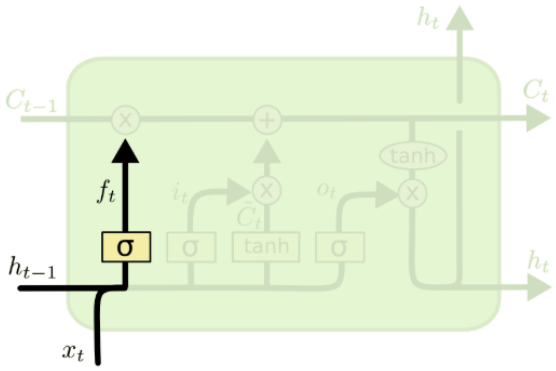


The repeating module in an LSTM contains four interacting layers.

5. LSTM

The structure of an LSTM network remains the same as an RNN, whereas the repeating module does more operations. Enhancing the repeating module enables the LSTM network to remember long-term dependencies. Let's try to break down each operation which helps the network remember better.

1. Forget gate operation

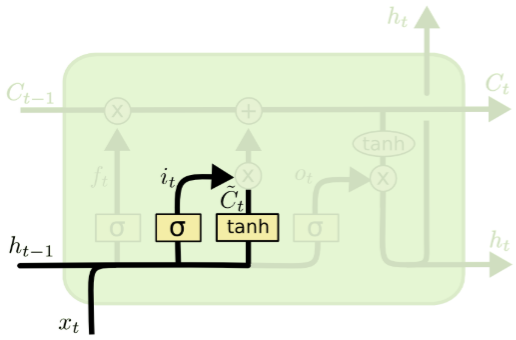


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

5. Forget operation

We take the input from current time step and the learned representation from previous time step and concatenate them. We pass the concatenated value into a sigmoid function which outputs a value(f_t) between 0 and 1. We do an element-wise multiplication between f_t and c_t-1. If a value is 0, then it is eliminated from c_t-1, if the value is 1, then it is completely let through. Therefore, this operation is also called "Forget gate operation".

2. Update gate operation

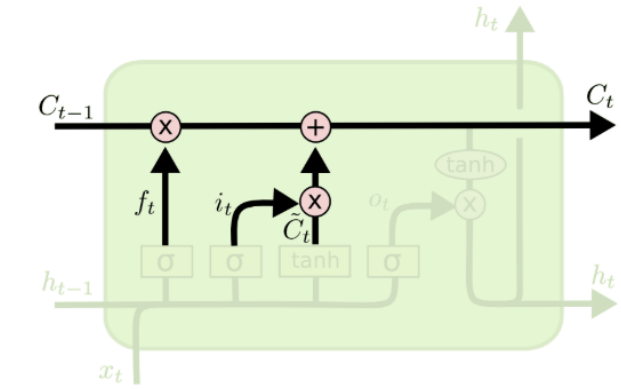


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

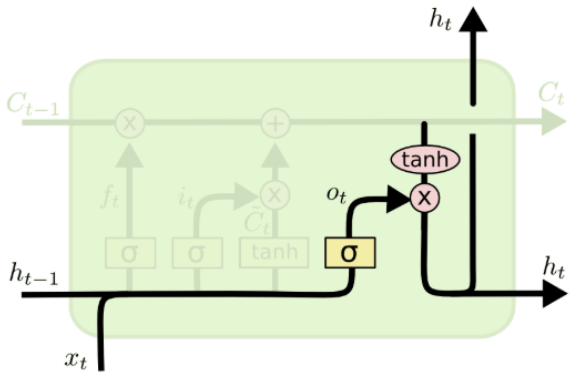
6. Update operation

The above figure represents the "Update gate operation". We concatenate values from current time step and the learned representation from previous time step. By passing the concatenated values through a tanh function we generate candidate values and by passing it through a sigmoid function we choose which values to be selected from the candidates. The chosen candidate values are updated to c_t-1.

3. Output gate operation



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$



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

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

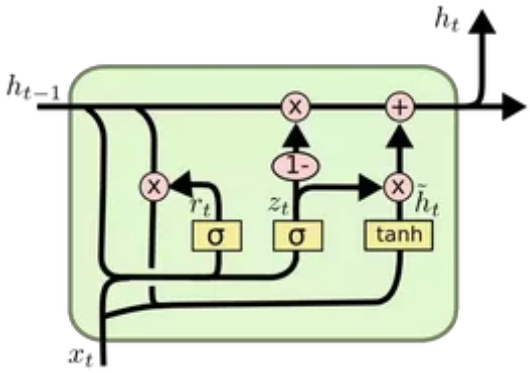
7. Updating the values(left) and Output operation(right)

We concatenate values from current time step and the learned representation from previous time step and pass it through a sigmoid function to choose which values we are going to use as the output. We take the cell state and apply a tanh function and do an element-wise operation which lets through only the selected outputs.

Now, this is a lot of operations to be done in a single cell. When using a bigger network, the training time would significantly increase compared to an RNN. There is an alternative to LSTM if want to reduce your training time but also use a network that remembers long term dependencies. It is called Gated Recurrent Unit(GRU).

Gated Recurrent Unit—GRU Network

A GRU unlike an LSTM network does not have a cell state and has 2 gates instead of 3(forget, update, output).



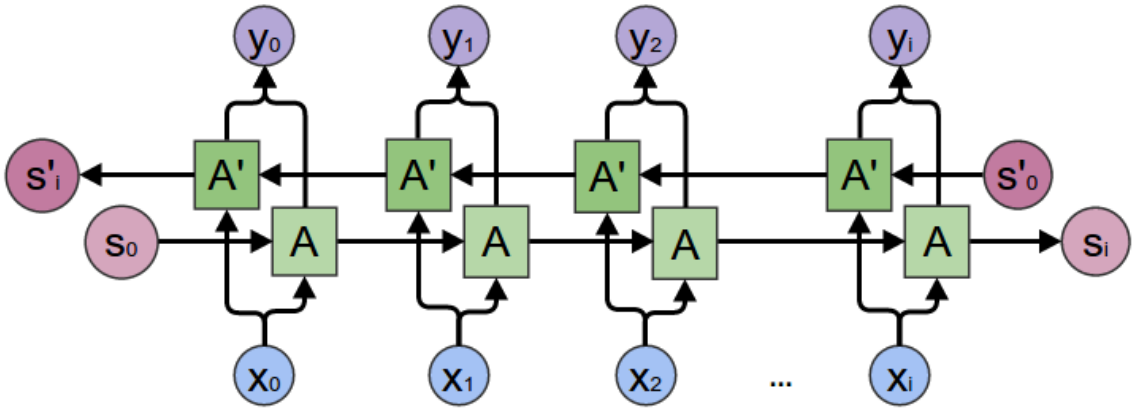
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

8. Gated recurrent Unit

A gated recurrent unit uses an update gate and a reset gate. The update gate decides on how much of information from the past should be let through and the reset gate decides on how much of information from the past should be discarded. In the above figure z_t represents the update gate operation, whereby using a sigmoid function, we decide on what values to let through from the past. h_t represents the reset gate operation, where we multiply the concatenated values from the previous time step and current time step with r_t. This produces the values that we would like to discard from the previous time steps.

Even though GRU is computationally efficient than an LSTM network, due to the reduction of gates, it still comes second to LSTM network in terms of performance. Therefore, GRU can be used when we need to train faster and don't have much computation power at hand.

Bidirectional RNN



9. Bidirectional RNN

A major issue with all of the above networks is that they learn representations from previous time steps. Sometimes, you might have to learn representations from future time steps to better understand the context and eliminate ambiguity. Take the following examples, "He said, Teddy bears are on sale" and "He said, Teddy Roosevelt was a great President". In the above two sentences, when we are looking at the word "Teddy" and the previous two words "He said", we might not be able to understand if the sentence refers to the President or Teddy bears. Therefore, to resolve this ambiguity, we need to look ahead. This is what Bidirectional RNNs accomplish.

The repeating module in a Bidirectional RNN could be a conventional RNN, LSTM or GRU. The structure and the connections of a bidirectional RNN are represented in figure 9. There are two type of connections, one going forward in time, which helps us learn from previous representations and another going backwards in time, which helps us learn from future representations.

Forward propagation is done in two steps:

- We move from left to right, starting with the initial time step we compute the values until we reach the final time step
- We move from right to left, starting with the final time step we compute the values until we reach the initial time step

Conclusion

Combining Bidirectional RNN with LSTM modules can significantly improve your performance and when you conflate them with an attention mechanism, you get state of the art performance in use cases such as machine translation, sentiment analysis etc. I hope this article was useful. There were a lot of mathematical equations involved and I hope it was not too intimidating. Let me know if have any doubts, I will try my best to sort it out :)

---

https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks/

When to Use MLP, CNN, and RNN Neural Networks

By Jason Brownlee on July 23, 2018 in Deep Learning

What neural network is appropriate for your predictive modeling problem?

It can be difficult for a beginner to the field of deep learning to know what type of network to use. There are so many types of networks to choose from and new methods being published and discussed every day.

To make things worse, most neural networks are flexible enough that they work (make a prediction) even when used with the wrong type of data or prediction problem.

In this post, you will discover the suggested use for the three main classes of artificial neural networks.

After reading this post, you will know:

- Which types of neural networks to focus on when working on a predictive modeling problem.
- When to use, not use, and possible try using an MLP, CNN, and RNN on a project.
- To consider the use of hybrid models and to have a clear idea of your project goals before selecting a model.

Let's get started.

When to Use MLP, CNN, and RNN Neural Networks
Photo by PRODAVID S. FERRY III,DDS, some rights reserved.

Overview

This post is divided into five sections; they are:

1. What Neural Networks to Focus on?
2. When to Use Multilayer Perceptrons?
3. When to Use Convolutional Neural Networks?
4. When to Use Recurrent Neural Networks?
5. Hybrid Network Models

What Neural Networks to Focus on?

Deep learning is the application of artificial neural networks using modern hardware.

It allows the development, training, and use of neural networks that are much larger (more layers) than was previously thought possible.

There are thousands of types of specific neural networks proposed by researchers as modifications or tweaks to existing models. Sometimes wholly new approaches.

As a practitioner, I recommend waiting until a model emerges as generally applicable. It is hard to tease out the signal of what works well generally from the noise of the vast number of publications released daily or weekly.

There are three classes of artificial neural networks that I recommend that you focus on in general. They are:

- Multilayer Perceptrons (MLPs)
- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs)

These three classes of networks provide a lot of flexibility and have proven themselves over decades to be useful and reliable in a wide range of problems. They also have many subtypes to help specialize them to the quirks of different framings of prediction problems and different datasets.

Now that we know what networks to focus on, let's look at when we can use each class of neural network.

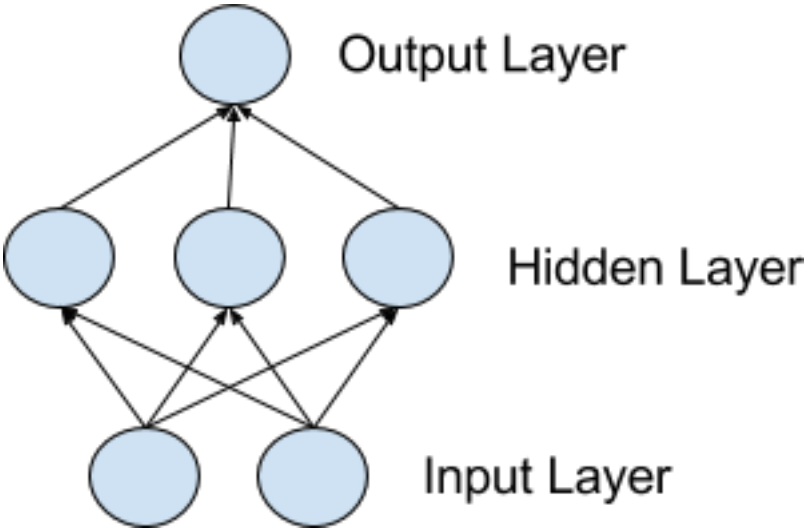
When to Use Multilayer Perceptrons?

Multilayer Perceptrons, or MLPs for short, are the classical type of neural network.

They are comprised of one or more layers of neurons. Data is fed to the input layer, there may be one or more hidden layers providing levels of abstraction, and predictions are made on the output layer, also called the visible layer.

For more details on the MLP, see the post:

- Crash Course On Multi-Layer Perceptron Neural Networks



Model of a Simple Network

MLPs are suitable for classification prediction problems where inputs are assigned a class or label.

They are also suitable for regression prediction problems where a real-valued quantity is predicted given a set of inputs. Data is often provided in a tabular format, such as you would see in a CSV file or a spreadsheet.

Use MLPs For:

- Tabular datasets
- Classification prediction problems
- Regression prediction problems

They are very flexible and can be used generally to learn a mapping from inputs to outputs.

This flexibility allows them to be applied to other types of data. For example, the pixels of an image can be reduced down to one long row of data and fed into a MLP. The words of a document can also be reduced to one long row of data and fed to a MLP. Even the lag observations for a time series prediction problem can be reduced to a long row of data and fed to a MLP.

As such, if your data is in a form other than a tabular dataset, such as an image, document, or time series, I would recommend at least testing an MLP on your problem. The results can be used as a baseline point of comparison to confirm that other models that may appear better suited add value.

Try MLPs On:

- Image data
- Text Data
- Time series data
- Other types of data

When to Use Convolutional Neural Networks?

Convolutional Neural Networks, or CNNs, were designed to map image data to an output variable.

They have proven so effective that they are the go-to method for any type of prediction problem involving image data as an input.

For more details on CNNs, see the post:

- Crash Course in Convolutional Neural Networks for Machine Learning

The benefit of using CNNs is their ability to develop an internal representation of a two-dimensional image. This allows the model to learn position and scale in variant structures in the data, which is important when working with images.

Use CNNs For:

- Image data
- Classification prediction problems
- Regression prediction problems

More generally, CNNs work well with data that has a spatial relationship.

The CNN input is traditionally two-dimensional, a field or matrix, but can also be changed to be one-dimensional, allowing it to develop an internal representation of a one-dimensional sequence.

This allows the CNN to be used more generally on other types of data that has a spatial relationship. For example, there is an order relationship between words in a document of text. There is an ordered relationship in the time steps of a time series.

Although not specifically developed for non-image data, CNNs achieve state-of-the-art results on problems such as document classification used in sentiment analysis and related problems.

Try CNNs On:

- Text data
- Time series data
- Sequence input data

When to Use Recurrent Neural Networks?

Recurrent Neural Networks, or RNNs, were designed to work with sequence prediction problems.

Sequence prediction problems come in many forms and are best described by the types of inputs and outputs supported.

Some examples of sequence prediction problems include:

- **One-to-Many**: An observation as input mapped to a sequence with multiple steps as an output.
- **Many-to-One**: A sequence of multiple steps as input mapped to class or quantity prediction.
- **Many-to-Many**: A sequence of multiple steps as input mapped to a sequence with multiple steps as output.

The Many-to-Many problem is often referred to as sequence-to-sequence, or seq2seq for short.

For more details on the types of sequence prediction problems, see the post:

- Gentle Introduction to Models for Sequence Prediction with Recurrent Neural Networks

Recurrent neural networks were traditionally difficult to train.

The Long Short-Term Memory, or LSTM, network is perhaps the most successful RNN because it overcomes the problems of training a recurrent network and in turn has been used on a wide range of applications.

For more details on RNNs, see the post:

- Crash Course in Recurrent Neural Networks for Deep Learning

RNNs in general and LSTMs in particular have received the most success when working with sequences of words and paragraphs, generally called natural language processing.

This includes both sequences of text and sequences of spoken language represented as a time series. They are also used as generative models that require a sequence output, not only with text, but on applications such as generating handwriting.

Use RNNs For:

- Text data
- Speech data
- Classification prediction problems
- Regression prediction problems
- Generative models

Recurrent neural networks are not appropriate for tabular datasets as you would see in a CSV file or spreadsheet. They are also not appropriate for image data input.

Don't Use RNNs For:

- Tabular data
- Image data

RNNs and LSTMs have been tested on time series forecasting problems, but the results have been poor, to say the least. Autoregression methods, even linear methods often perform much better. LSTMs are often outperformed by simple MLPs applied on the same data.

For more on this topic, see the post:

- On the Suitability of Long Short-Term Memory Networks for Time Series Forecasting

Nevertheless, it remains an active area.

Perhaps Try RNNs on:

- Time series data

Hybrid Network Models

A CNN or RNN model is rarely used alone.

These types of networks are used as layers in a broader model that also has one or more MLP layers. Technically, these are a hybrid type of neural network architecture.

Perhaps the most interesting work comes from the mixing of the different types of networks together into hybrid models.

For example, consider a model that uses a stack of layers with a CNN on the input, LSTM in the middle, and MLP at the output. A model like this can read a sequence of image inputs, such as a video, and generate a prediction. This is called a CNN LSTM architecture.

The network types can also be stacked in specific architectures to unlock new capabilities, such as the reusable image recognition models that use very deep CNN and MLP networks that can be added to a new LSTM model and used for captioning photos. Also, the encoder-decoder LSTM networks that can be used to have input and output sequences of differing lengths.

It is important to think clearly about what you and your stakeholders require from the project first, then seek out a network architecture (or develop one) that meets your specific project needs.

---

https://machinelearningmastery.com/crash-course-convolutional-neural-networks/

Crash Course in Convolutional Neural Networks for Machine Learning

By Jason Brownlee on June 24, 2016 in Deep Learning

Convolutional Neural Networks are a powerful artificial neural network technique.

These networks preserve the spatial structure of the problem and were developed for object recognition tasks such as handwritten digit recognition. They are popular because people are achieving state-of-the-art results on difficult computer vision and natural language processing tasks.

In this post you will discover Convolutional Neural Networks for deep learning, also called ConvNets or CNNs. After completing this crash course you will know:

- The building blocks used in CNNs such as convolutional layers and pool layers.
- How the building blocks fit together with a short worked example.
- Best practices for configuring CNNs on your own object recognition tasks.
- References for state of the art networks applied to complex machine learning problems.

Let's get started.

Crash Course in Convolutional Neural Networks for Machine Learning
Photo by Bryan Ledgard, some rights reserved.

The Case for Convolutional Neural Networks

Given a dataset of gray scale images with the standardized size of 32×32 pixels each, a traditional feedforward neural network would require 1024 input weights (plus one bias).

This is fair enough, but the flattening of the image matrix of pixels to a long vector of pixel values loses all of the spatial structure in the image. Unless all of the images are perfectly resized, the neural network will have great difficulty with the problem.

Convolutional Neural Networks expect and preserve the spatial relationship between pixels by learning internal feature representations using small squares of input data. Feature are learned and used across the whole image, allowing for the objects in the images to be shifted or translated in the scene and still detectable by the network.

It is this reason why the network is so useful for object recognition in photographs, picking out digits, faces, objects and so on with varying orientation.

In summary, below are some benefits of using convolutional neural networks:

- They use fewer parameters (weights) to learn than a fully connected network.
- They are designed to be invariant to object position and distortion in the scene.
- They automatically learn and generalize features from the input domain.

Building Blocks of Convolutional Neural Networks

There are three types of layers in a Convolutional Neural Network:

1. Convolutional Layers.
2. Pooling Layers.
3. Fully-Connected Layers.

1. Convolutional Layers

Convolutional layers are comprised of filters and feature maps.

Filters

The filters are the "neurons" of the layer. The have input weights and output a value. The input size is a fixed square called a patch or a receptive field.

If the convolutional layer is an input layer, then the input patch will be pixel values. If the deeper in the network architecture, then the convolutional layer will take input from a feature map from the previous layer.

Feature Maps

The feature map is the output of one filter applied to the previous layer.

A given filter is drawn across the entire previous layer, moved one pixel at a time. Each position results in an activation of the neuron and the output is collected in the feature map. You can see that if the receptive field is moved one pixel from activation to activation, then the field will overlap with the previous activation by (field width – 1) input values.

Zero Padding

The distance that filter is moved across the the input from the previous layer each activation is referred to as the stride.

If the size of the previous layer is not cleanly divisible by the size of the filters receptive field and the size of the stride then it is possible for the receptive field to attempt to read off the edge of the input feature map. In this case, techniques like zero padding can be used to invent mock inputs for the receptive field to read.

2. Pooling Layers

The pooling layers down-sample the previous layers feature map.

Pooling layers follow a sequence of one or more convolutional layers and are intended to consolidate the features learned and expressed in the previous layers feature map. As such, pooling may be consider a technique to compress or generalize feature representations and generally reduce the overfitting of the training data by the model.

They too have a receptive field, often much smaller than the convolutional layer. Also, the stride or number of inputs that the receptive field is moved for each activation is often equal to the size of the receptive field to avoid any overlap.

Pooling layers are often very simple, taking the average or the maximum of the input value in order to create its own feature map.

3. Fully Connected Layers

Fully connected layers are the normal flat feed-forward neural network layer.

These layers may have a non-linear activation function or a softmax activation in order to output probabilities of class predictions.

Fully connected layers are used at the end of the network after feature extraction and consolidation has been performed by the convolutional and pooling layers. They are used to create final non-linear combinations of features and for making predictions by the network.

Worked Example of a Convolutional Neural Network

You now know about convolutional, pooling and fully connected layers. Let’s make this more concrete by working through how these three layers may be connected together.

1. Image Input Data

Let's assume we have a dataset of grayscale images. Each image has the same size of 32 pixels wide and 32 pixels high, and pixel values are between 0 and 255, g.e. a matrix of 32x32x1 or 1024 pixel values.

Image input data is expressed as a 3-dimensional matrix of width * height * channels. If we were using color images in our example, we would have 3 channels for the red, green and blue pixel values, e.g. 32x32x3.

2. Convolutional Layer

We define a convolutional layer with 10 filters and a receptive field 5 pixels wide and 5 pixels high and a stride length of 1.

Because each filter can only get input from (i.e. "see") 5×5 (25) pixels at a time, we can calculate that each will require 25 + 1 input weights (plus 1 for the bias input).

Dragging the 5×5 receptive field across the input image data with a stride width of 1 will result in a feature map of 28×28 output values or 784 distinct activations per image.

We have 10 filters, so that is 10 different 28×28 feature maps or 7,840 outputs that will be created for one image.

Finally, we know we have 26 inputs per filter, 10 filters and 28×28 output values to calculate per filter, therefore we have a total of 26x10x28x28 or 203,840 "connections" in our convolutional layer, we we want to phrase it using traditional neural network nomenclature.

Convolutional layers also make use of a nonlinear transfer function as part of activation and the rectifier activation function is the popular default to use.

3. Pool Layer

We define a pooling layer with a receptive field with a width of 2 inputs and a height of 2 inputs. We also use a stride of 2 to ensure that there is no overlap.

This results in feature maps that are one half the size of the input feature maps. From 10 different 28×28 feature maps as input to 10 different 14×14 feature maps as output.

We will use a max() operation for each receptive field so that the activation is the maximum input value.

4. Fully Connected Layer

Finally, we can flatten out the square feature maps into a traditional flat fully connected layer.

We can define the fully connected layer with 200 hidden neurons, each with 10x14x14 input connections, or 1960 + 1 weights per neuron. That is a total of 392,200 connections and weights to learn in this layer.

We can use a sigmoid or softmax transfer function to output probabilities of class values directly.

Convolutional Neural Networks Best Practices

Now that we know about the building blocks for a convolutional neural network and how the layers hang together, we can review some best practices to consider when applying them.

- **Input Receptive Field Dimensions**: The default is 2D for images, but could be 1D such as for words in a sentence or 3D for video that adds a time dimension.
- **Receptive Field Size**: The patch should be as small as possible, but large enough to "see" features in the input data. It is common to use 3×3 on small images and 5×5 or 7×7 and more on larger image sizes.
- **Stride Width**: Use the default stride of 1. It is easy to understand and you don't need padding to handle the receptive field falling off the edge of your images. This could increased to 2 or larger for larger images.
- **Number of Filters**: Filters are the feature detectors. Generally fewer filters are used at the input layer and increasingly more filters used at deeper layers.
- **Padding**: Set to zero and called zero padding when reading non-input data. This is useful when you cannot or do not want to standardize input image sizes or when you want to use receptive field and stride sizes that do not neatly divide up the input image size.
- **Pooling**: Pooling is a destructive or generalization process to reduce overfitting. Receptive field is almost always set to to 2×2 with a stride of 2 to discard 75% of the activations from the output of the previous layer.
- **Data Preparation**: Consider standardizing input data, both the dimensions of the images and pixel values.
- **Pattern Architecture**: It is common to pattern the layers in your network architecture. This might be one, two or some number of convolutional layers followed by a pooling layer. This structure can then be repeated one or more times. Finally, fully connected layers are often only used at the output end and may be stacked one, two or more deep.
- **Dropout**: CNNs have a habit of overfitting, even with pooling layers. Dropout should be used such as between fully connected layers and perhaps after pooling layers.

Do you know about some more best practices for using CNNs?

Let me know in the comments.

Further Reading on Convolutional Neural Networks

You have only scratch the surface on convolutional neural networks. The field is moving very fast and new and interesting architectures and techniques are been discussed and used all the time.

If you are looking for a deeper understanding of the technique, take a look at LeCun, et. al's seminal paper titled "Gradient-Based Learning Applied to Document Recognition" [PDF]. In

it they introduce LeNet applied to handwritten digit recognition and carefully explain the layers and how the network is connected.

There are a lot of tutorials and discussions of CNNs around the web. A few choice examples are listed below. Personally I find the explanatory pictures in the posts useful only after understanding how the network hangs together, many of the explanations are confusing and defer you to LeCun's paper if in doubt.

- Convolutional Networks in DeepLearning4J
- Convolutional Networks model in the Stanford CS231n course
- Convolutional Networks and Applications in Vision [PDF]
- Chapter 6 in Michael Nielsen's open Deep Learning book
- VGG Convolutional Neural Networks Practical from Oxford
- Understanding Convolutional Neural Networks for NLP by Denny Britz

Summary

In this post you discovered convolutional neural networks. You learned about:

- Why CNNs are needed to preserve spatial structure in your input data and the benefits they provide.
- The building blocks of CNN including convolutional, pooling and fully connected layers.
- How the layers in a CNN hang together.
- Best practices when applying CNN to your own problems.

Do you have any questions about convolutional neural networks or about this post? Ask your questions in the comments and I will do my best to answer.

---

https://machinelearningmastery.com/crash-course-recurrent-neural-networks-deep-learning/

Crash Course in Recurrent Neural Networks for Deep Learning

By Jason Brownlee on July 7, 2016 in Long Short-Term Memory Networks

There is another type of neural network that is dominating difficult machine learning problems that involve sequences of inputs called recurrent neural networks.

Recurrent neural networks have connections that have loops, adding feedback and memory to the networks over time. This memory allows this type of network to learn and generalize across sequences of inputs rather than individual patterns.

A powerful type of Recurrent Neural Network called the Long Short-Term Memory Network has been shown to be particularly effective when stacked into a deep configuration, achieving state-of-the-art results on a diverse array of problems from language translation to automatic captioning of images and videos.

In this post you will get a crash course in recurrent neural networks for deep learning, acquiring just enough understanding to start using LSTM networks in Python with Keras.

After reading this post, you will know:

- The limitations of Multilayer Perceptrons that are addressed by recurrent neural networks.
- The problems that must be addressed to make Recurrent Neural networks useful.
- The details of the Long Short-Term Memory networks used in applied deep learning.

Let's get started.

Crash Course in Recurrent Neural Networks for Deep Learning
Photo by Martin Fisch, some rights reserved.

Support For Sequences in Neural Networks

There are some problem types that are best framed involving either a sequence as an input or an output.

For example, consider a univariate time series problem, like the price of a stock over time. This dataset can be framed as a prediction problem for a classical feedforward multilayer Perceptron network by defining a windows size (e.g. 5) and training the network to learn to make short term predictions from the fixed sized window of inputs.

This would work, but is very limited. The window of inputs adds memory to the problem, but is limited to just a fixed number of points and must be chosen with sufficient knowledge of

the problem. A naive window would not capture the broader trends over minutes, hours and days that might be relevant to making a prediction. From one prediction to the next, the network only knows about the specific inputs it is provided.

Univariate time series prediction is important, but there are even more interesting problems that involve sequences.

Consider the following taxonomy of sequence problems that require a mapping of an input to an output (taken from Andrej Karpathy).

- **One-to-Many**: sequence output, for image captioning.
- **Many-to-One**: sequence input, for sentiment classification.
- **Many-to-Many**: sequence in and out, for machine translation.
- **Synched Many-to-Many**: synced sequences in and out, for video classification.

We can also see that a one-to-one example of input to output would be an example of a classical feedforward neural network for a prediction task like image classification.

Support for sequences in neural networks is an important class of problem and one where deep learning has recently shown impressive results State-of-the art results have been using a type of network specifically designed for sequence problems called recurrent neural networks.

Recurrent Neural Networks

Recurrent Neural Networks or RNNs are a special type of neural network designed for sequence problems.

Given a standard feed-forward multilayer Perceptron network, a recurrent neural network can be thought of as the addition of loops to the architecture. For example, in a given layer, each neuron may pass its signal latterly (sideways) in addition to forward to the next layer. The output of the network may feedback as an input to the network with the next input vector. And so on.

The recurrent connections add state or memory to the network and allow it to learn broader abstractions from the input sequences.

The field of recurrent neural networks is well established with popular methods. For the techniques to be effective on real problems, two major issues needed to be resolved for the network to be useful.

1. How to train the network with Backpropagation.

2. How to stop gradients vanishing or exploding during training.

1. How to Train Recurrent Neural Networks

The staple technique for training feedforward neural networks is to back propagate error and update the network weights.

Backpropagation breaks down in a recurrent neural network, because of the recurrent or loop connections.

This was addressed with a modification of the Backpropagation technique called Backpropagation Through Time or BPTT.

Instead of performing backpropagation on the recurrent network as stated, the structure of the network is unrolled, where copies of the neurons that have recurrent connections are created. For example a single neuron with a connection to itself (A->A) could be represented as two neurons with the same weight values (A->B).

This allows the cyclic graph of a recurrent neural network to be turned into an acyclic graph like a classic feed-forward neural network, and Backpropagation can be applied.

2. How to Have Stable Gradients During Training

When Backpropagation is used in very deep neural networks and in unrolled recurrent neural networks, the gradients that are calculated in order to update the weights can become unstable.

They can become very large numbers called exploding gradients or very small numbers called the vanishing gradient problem. These large numbers in turn are used to update the weights in the network, making training unstable and the network unreliable.

This problem is alleviated in deep multilayer Perceptron networks through the use of the Rectifier transfer function, and even more exotic but now less popular approaches of using unsupervised pre-training of layers.

In recurrent neural network architectures, this problem has been alleviated using a new type of architecture called the Long Short-Term Memory Networks that allows deep recurrent networks to be trained.

Long Short-Term Memory Networks

The Long Short-Term Memory or LSTM network is a recurrent neural network that is trained using Backpropagation Through Time and overcomes the vanishing gradient problem.

As such it can be used to create large (stacked) recurrent networks, that in turn can be used to address difficult sequence problems in machine learning and achieve state-of-the-art results.

Instead of neurons, LSTM networks have memory blocks that are connected into layers.

A block has components that make it smarter than a classical neuron and a memory for recent sequences. A block contains gates that manage the block's state and output. A unit operates upon an input sequence and each gate within a unit uses the sigmoid activation function to control whether they are triggered or not, making the change of state and addition of information flowing through the unit conditional.

There are three types of gates within a memory unit:

- **Forget Gate**: conditionally decides what information to discard from the unit.
- **Input Gate**: conditionally decides which values from the input to update the memory state.
- **Output Gate**: conditionally decides what to output based on input and the memory of the unit.

Each unit is like a mini state machine where the gates of the units have weights that are learned during the training procedure.

You can see how you may achieve a sophisticated learning and memory from a layer of LSTMs, and it is not hard to imagine how higher-order abstractions may be layered with multiple such layers.

Resources

We have covered a lot of ground in this post. Below are some resources that you can use to go deeper into the topic of recurrent neural networks for deep learning.

Resources to learn more about Recurrent Neural Networks and LSTMs.

- Recurrent neural network on Wikipedia
- Long Short-Term Memory on Wikipedia
- The Unreasonable Effectiveness of Recurrent Neural Networks by Andrej Karpathy
- Understanding LSTM Networks
- Deep Dive into Recurrent Neural Nets
- A Beginner's Guide to Recurrent Networks and LSTMs

Popular tutorials for implementing LSTMs.

- LSTMs for language modeling with TensorFlow
- RNN for Spoken Word Understanding in Theano
- LSTM for sentiment analysis in Theano

Primary sources on LSTMs.

- Long short-term memory [pdf], 1997 paper by Hochreiter and Schmidhuber
- Learning to forget: Continual prediction with LSTM, 2000 by Schmidhuber and Cummins that add the forget gate
- On the difficulty of training Recurrent Neural Networks [pdf], 2013

People to follow doing great work with LSTMs.

- Alex Graves
- Jürgen Schmidhuber
- Ilya Sutskever
- Tomas Mikolov

Summary

In this post you discovered sequence problems and recurrent neural networks that can be used to address them.

Specifically, you learned:

- The limitations of classical feedforward neural networks and how recurrent neural networks can overcome these problems.
- The practical problems in training recurrent neural networks and how they are overcome.
- The Long Short-Term Memory network used to create deep recurrent neural networks.

---

https://towardsdatascience.com/understanding-neural-networks-from-neuron-to-rnn-cnn-and-deep-learning-cd88e90e0a90

Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning

vibhor nigam
Sep 10, 2018

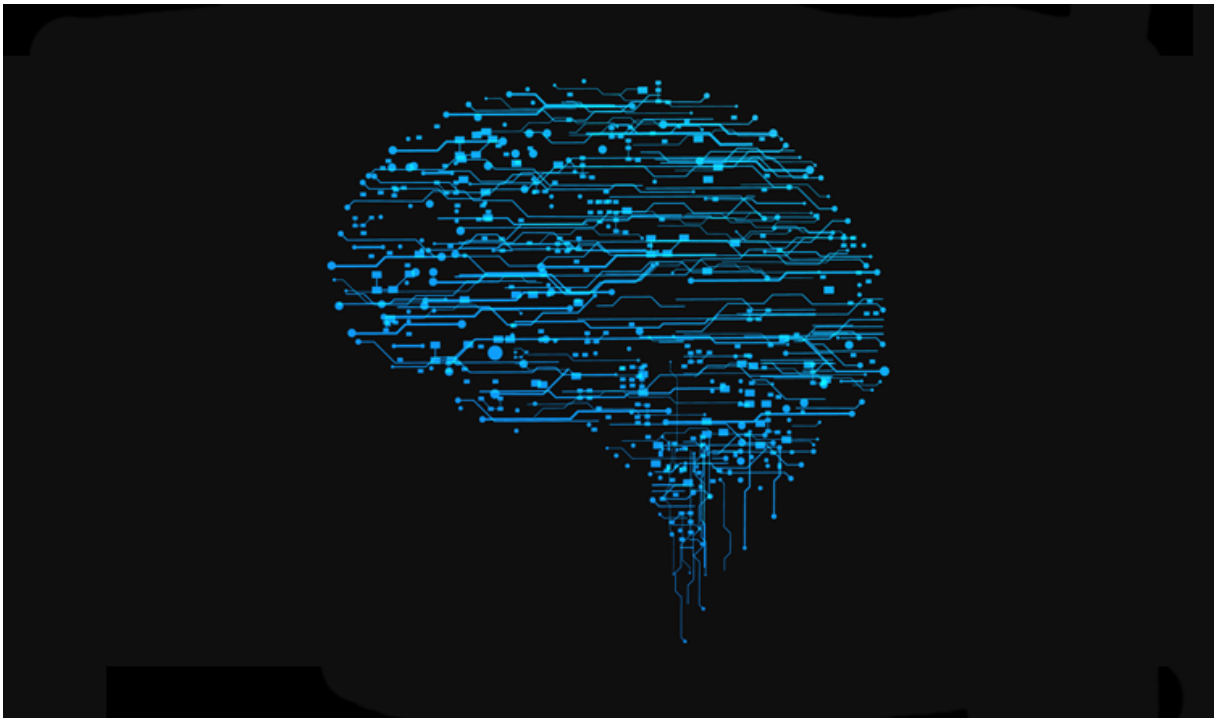
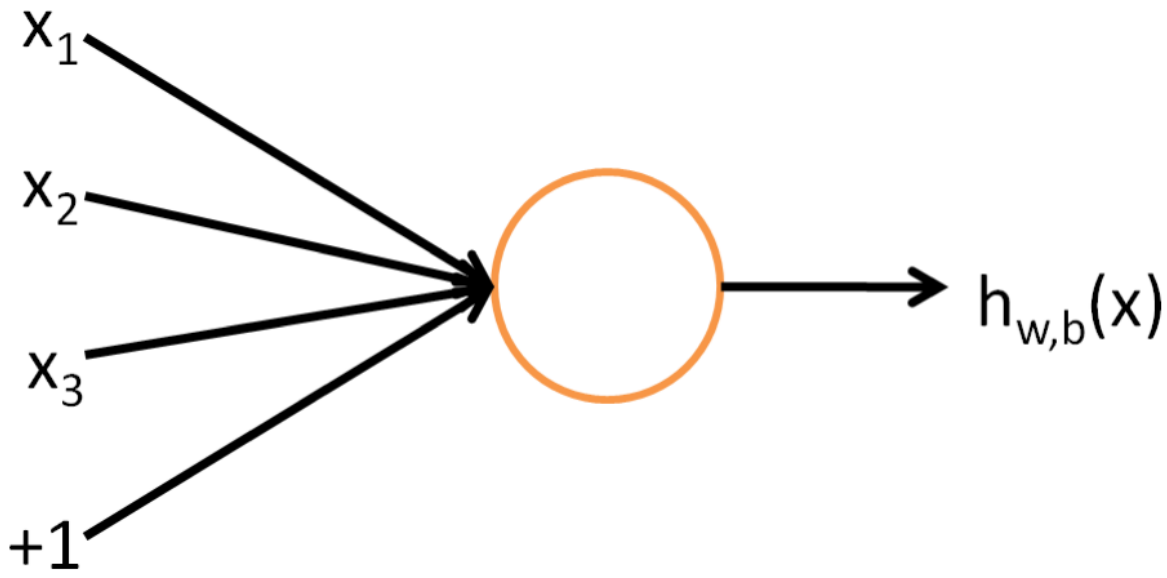


Image credit Google

Neural Networks is one of the most popular machine learning algorithms at present. It has been decisively proven over time that neural networks outperform other algorithms in accuracy and speed. With various variants like CNN (Convolutional Neural Networks), RNN(Recurrent Neural Networks), AutoEncoders, Deep Learning etc. neural networks are slowly becoming for data scientists or machine learning practitioners what linear regression was one for statisticians. It is thus imperative to have a fundamental understanding of what a Neural Network is, how it is made up and what is its reach and limitations. This post is an attempt to explain a neural network starting from its most basic building block a neuron, and later delving into its most popular variations like CNN, RNN etc.

What is a Neuron?

As the name suggests, neural networks were inspired by the neural architecture of a human brain, and like in a human brain the basic building block is called a Neuron. Its functionality is similar to a human neuron, i.e. it takes in some inputs and fires an output. In purely mathematical terms, a neuron in the machine learning world is a placeholder for a mathematical function, and its only job is to provide an output by applying the function on the inputs provided.

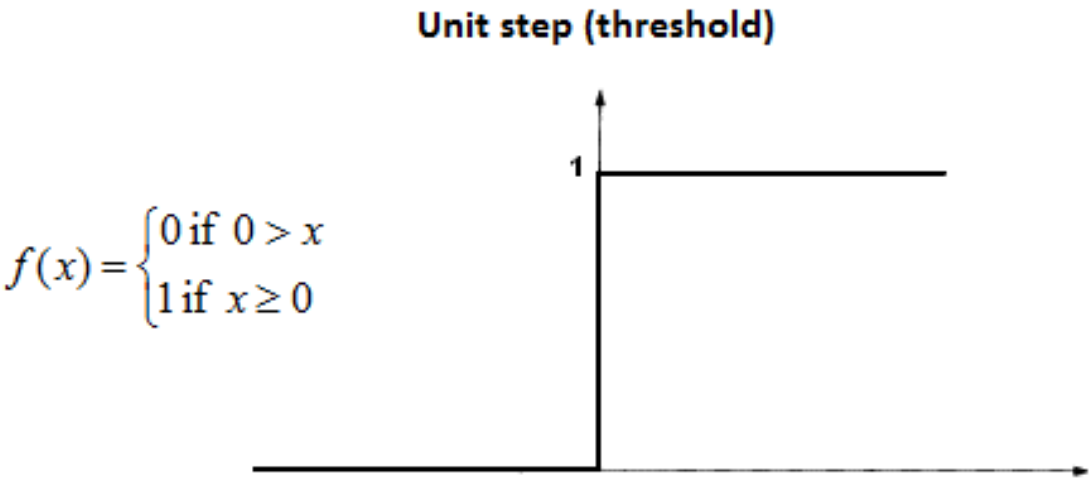


The function used in a neuron is generally termed as an activation function. There have been 5 major activation functions tried to date, step, sigmoid, tanh, and ReLU. Each of these is described in detail below.

ACTIVATION FUNCTIONS

Step function

A step function is defined as

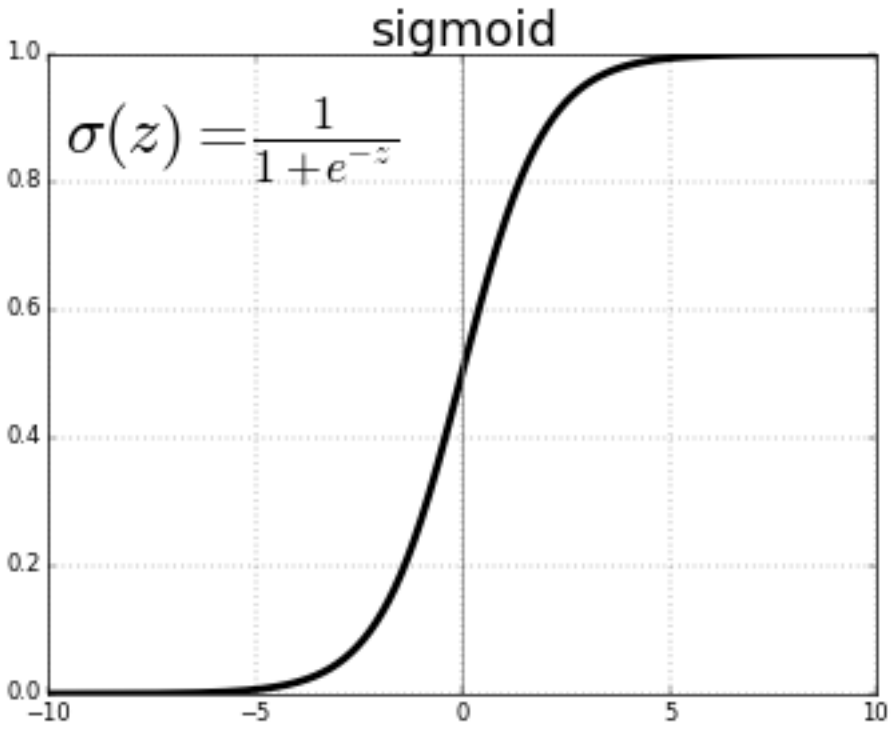


Where the output is 1 if the value of x is greater than equal to zero and 0 if the value of x is less than zero. As one can see a step function is non-differentiable at zero. At present, a neural network uses back propagation method along with gradient descent to calculate weights of different layers. Since the step function is non-differentiable at zero hence it is not able to make progress with the gradient descent approach and fails in the task of updating the weights.

To overcome, this problem sigmoid functions were introduced instead of the step function.

Sigmoid Function

A sigmoid function or logistic function is defined mathematically as



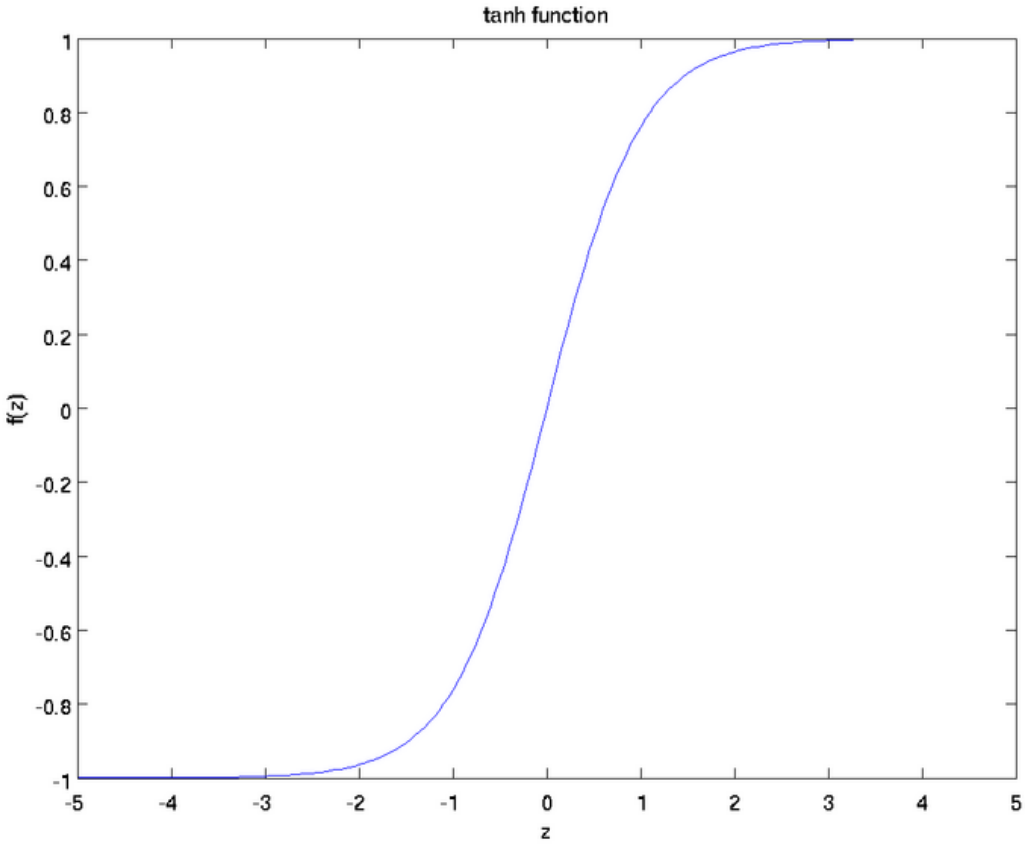
The value of the function tends to zero when z or independent variable tends to negative infinity and tends to 1 when z tends to infinity. It needs to be kept in mind that this function represents an approximation of the behavior of the dependent variable and is an assumption. Now the question arises as to why we use the sigmoid function as one of the approximation functions. There are certain simple reasons for this.

1. It captures non-linearity in the data. Albeit in an approximated form, but the concept of non-linearity is essential for accurate modeling.
2. The sigmoid function is differentiable throughout and hence can be used with gradient descent and backpropagation approaches for calculating weights of different layers
3. The assumption of a dependent variable to follow a sigmoid function inherently assumes a Gaussian distribution for the independent variable which is a general distribution we see for a lot of randomly occurring events and this is a good generic distribution to start with.

However, a sigmoid function also suffers from a problem of vanishing gradients. As can be seen from the picture a sigmoid function squashes it's input into a very small output range [0,1] and has very steep gradients. Thus, there remain large regions of input space, where even a large change produces a very small change in the output. This is referred to as the problem of vanishing gradient. This problem increases with an increase in the number of layers and thus stagnates the learning of a neural network at a certain level.

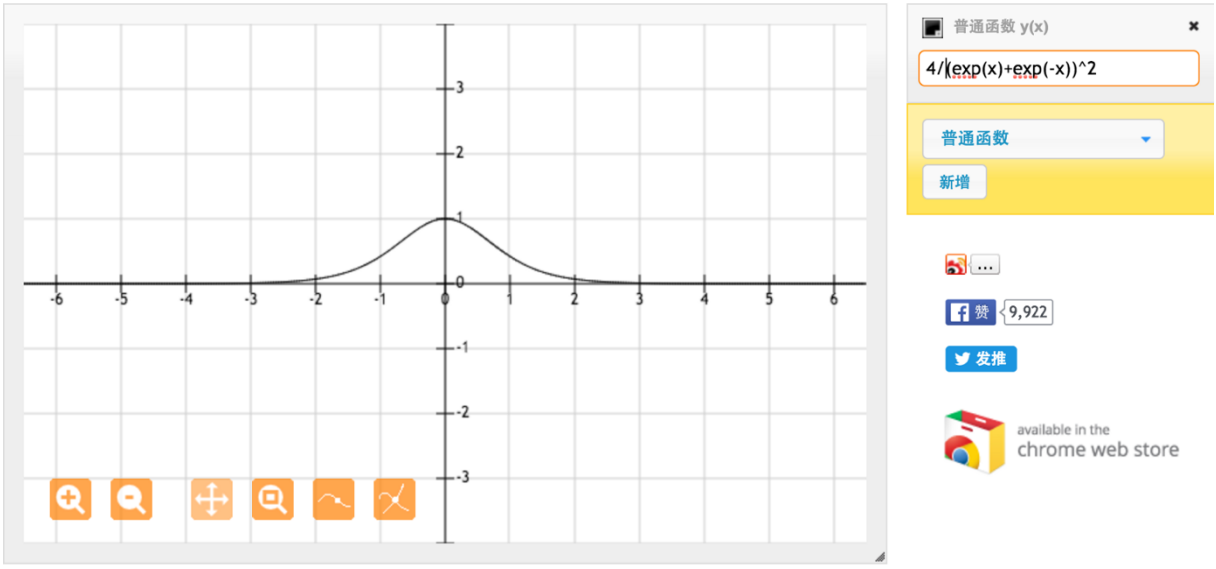
Tanh Function

The tanh(z) function is a rescaled version of the sigmoid, and its output range is [− 1,1] instead of [0,1]. [1]

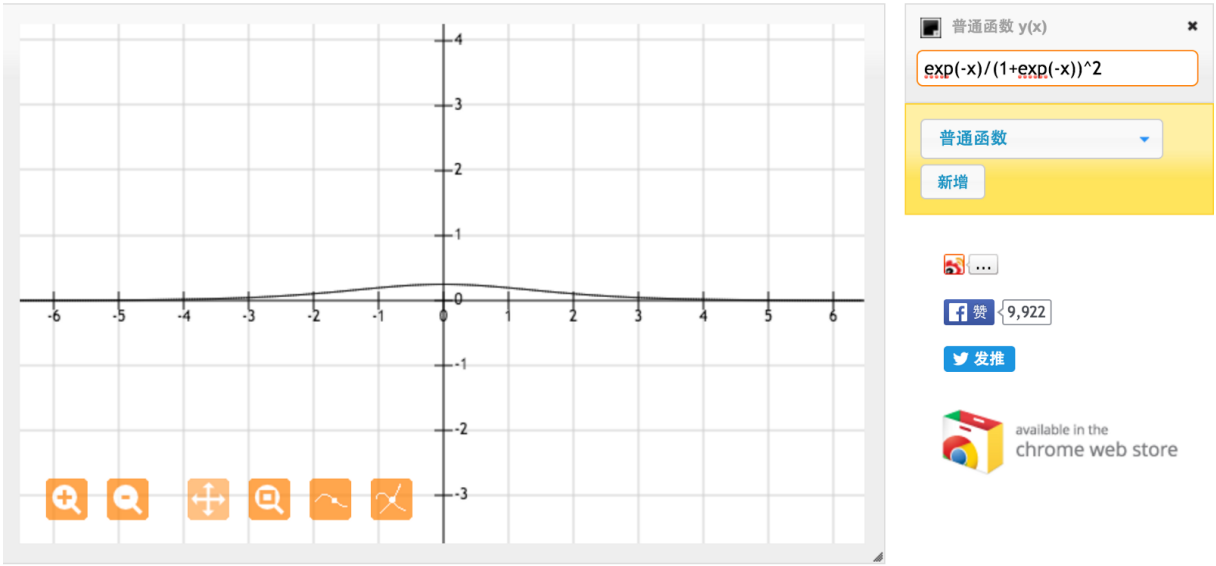


The general reason for using a Tanh function in some places instead of the sigmoid function is because since data is centered around 0, the derivatives are higher. A higher gradient helps in a better learning rate. Below attached are plotted gradients of two functions tanh and sigmoid. [2]

For tanh function, for an input between [-1,1], we have derivative between [0.42, 1].



For sigmoid function on the other hand, for input between [0,1], we have derivative between [0.20, 0.25]

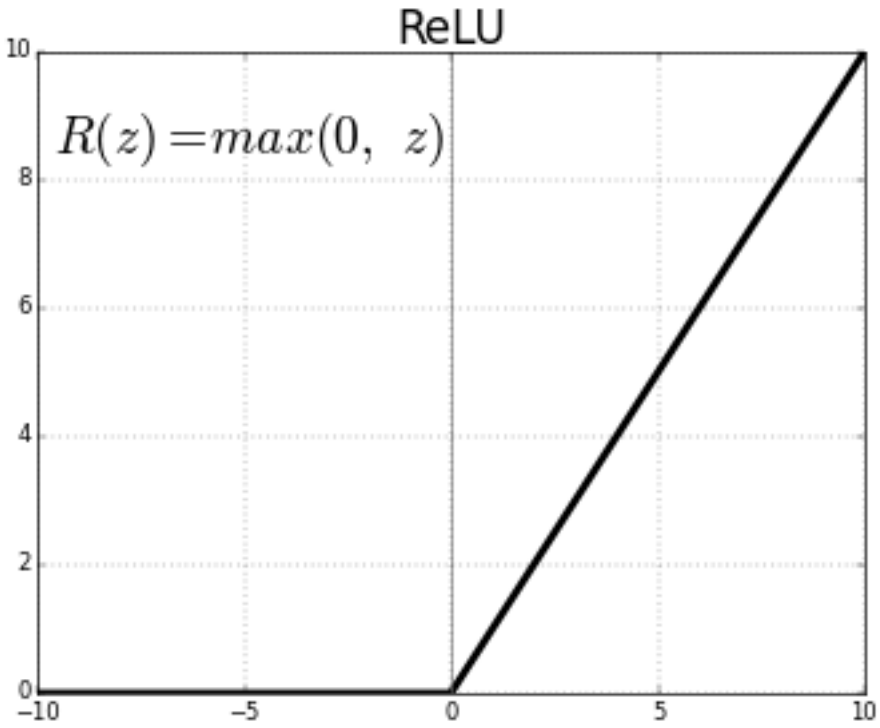


As one can see from the pictures above a Tanh function has a higher range of derivative than a Sigmoid function and thus has a better learning rate. However, the problem of vanishing gradients still persists in Tanh function.

ReLU Function

The Rectified Linear Unit is the most commonly used activation function in deep learning models. The function returns 0 if it receives any negative input, but for any positive value x, it returns that value back. So, it can be written as f(x)=max (0, x).

Graphically it looks like this [3]



The Leaky ReLU is one of the most well-known. It is the same as ReLU for positive numbers. But instead of being 0 for all negative values, it has a constant slope (less than 1.).

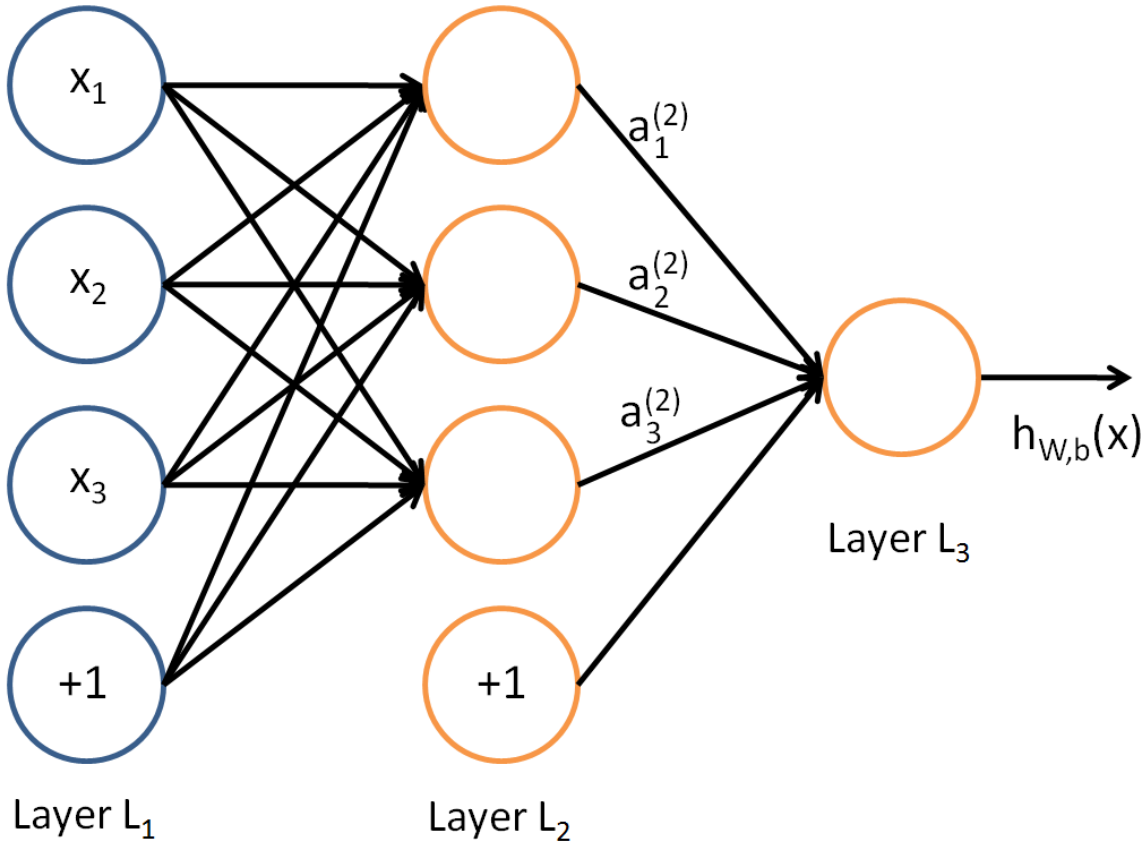
That slope is a parameter the user sets when building the model, and it is frequently called α. For example, if the user sets α=0.3, the activation function is.`f(x) = max (0.3*x, x)` This has the theoretical advantage that, by being influenced, by `x` at all values, it may make more complete use of the information contained in x.

There are other alternatives, but both practitioners and researchers have generally found an insufficient benefit to justify using anything other than ReLU. In general practice as well, ReLU has found to be performing better than sigmoid or tanh functions.

Neural Networks

Till now we have covered neuron and activation functions which together for the basic building blocks of any neural network. Now, we will dive in deeper into what is a Neural Network and different types of it. I would highly suggest people, to revisit neurons and activation functions if they have a doubt about it.

Before understanding a Neural Network, it is imperative to understand what is a layer in a Neural Network. A layer is nothing but a collection of neurons which take in an input and provide an output. Inputs to each of these neurons are processed through the activation functions assigned to the neurons. For example, here is a small neural network.



The leftmost layer of the network is called the **input layer**, and the rightmost layer the **output layer** (which, in this example, has only one node). The middle layer of nodes is called the **hidden layer** because its values are not observed in the training set. We also say that our example neural network has 3 **input units** (not counting the bias unit), 3 **hidden units**, and 1 **output unit** [4]

Any neural network has 1 input and 1 output layer. The number of hidden layers, for instance, differ between different networks depending upon the complexity of the problem to be solved.

Another important point to note here is that each of the hidden layers can have a different activation function, for instance, hidden layer1 may use a sigmoid function and hidden layer2 may use a ReLU, followed by a Tanh in hidden layer3 all in the same neural network. Choice of the activation function to be used again depends on the problem in question and the type of data being used.

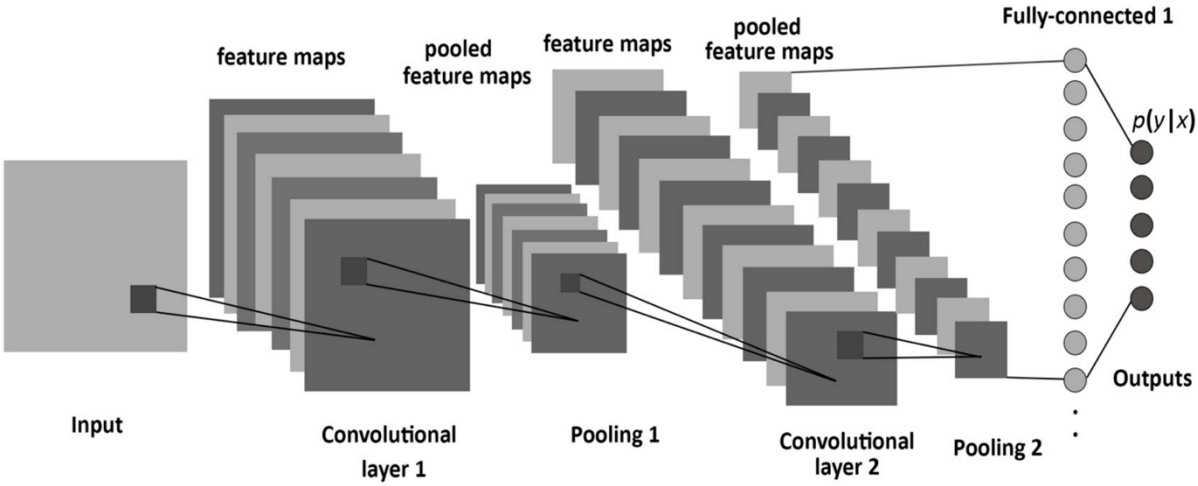
Now for a neural network to make accurate predictions each of these neurons learn certain weights at every layer. The algorithm through which they learn the weights is called back propagation, the details of which are beyond the scope of this post.

A neural network having more than one hidden layer is generally referred to as a Deep Neural Network.

Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) is one of the variants of neural networks used heavily in the field of Computer Vision. It derives its name from the type of hidden layers it consists of. The hidden layers of a CNN typically consist of convolutional layers, pooling layers, fully

connected layers, and normalization layers. Here it simply means that instead of using the normal activation functions defined above, convolution and pooling functions are used as activation functions.



To understand it in detail one needs to understand what convolution and pooling are. Both of these concepts are borrowed from the field of Computer Vision and are defined below.

Convolution: Convolution operates on two signals (in 1D) or two images (in 2D): you can think of one as the "input" signal (or image), and the other (called the kernel) as a "filter" on the input image, producing an output image (so convolution takes two images as input and produces a third as output). [5]
In layman terms it takes in an input signal and applies a filter over it, essentially multiplies the input signal with the kernel to get the modified signal. Mathematically, a convolution of two functions f and g is defined as

$$(f * g)(i) = \sum_{j=1}^{m} g(j) \cdot f(i - j + m/2)$$

which, is nothing but dot product of the input function and a kernel function.

In case of Image processing, it is easier to visualize a kernel as sliding over an entire image and thus changing the value of each pixel in the process.

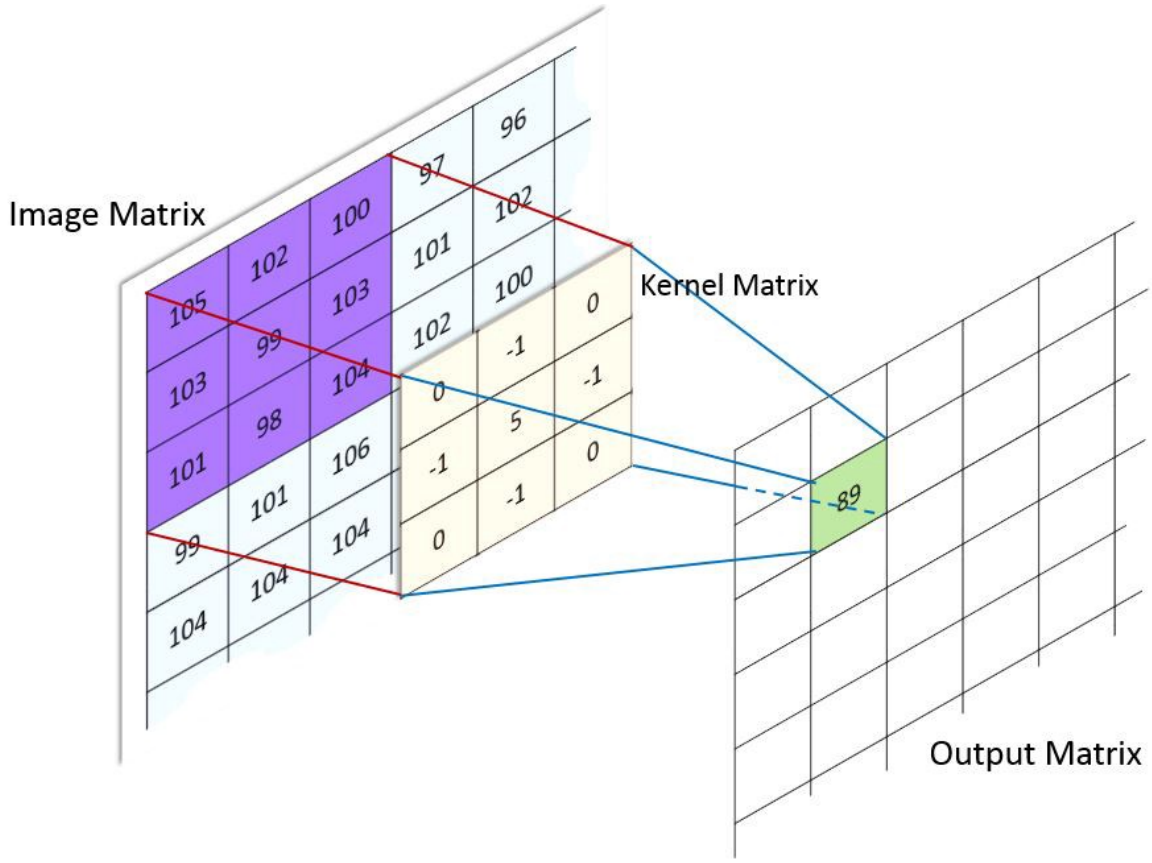
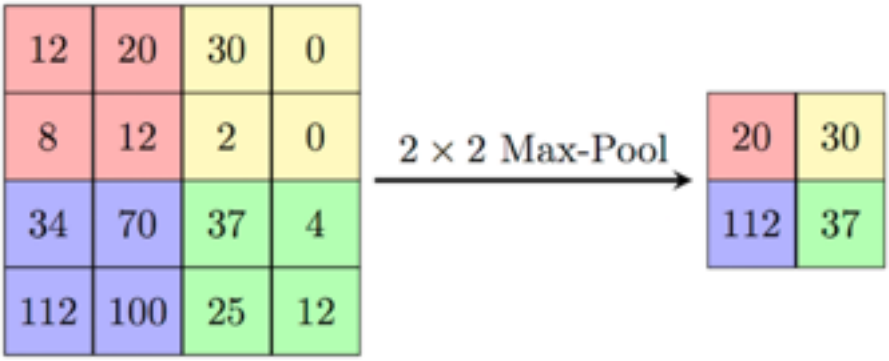


Image Credit: Machine Learning Guru [6]

Pooling: Pooling is a **sample-based discretization process**. The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned.

There are 2 main types of pooling commonly known as max and min pooling. As the name suggests max pooling is based on picking up the maximum value from the selected region and min pooling is based on picking up the minimum value from the selected region.



Thus as one can see A Convolutional Neural Network or CNN is basically a deep neural network which consists of hidden layers having convolution and pooling functions in addition to the activation function for introducing non-linearity.

A more detailed explanation can be found at

http://colah.github.io/posts/2014-07-Conv-Nets-Modular/

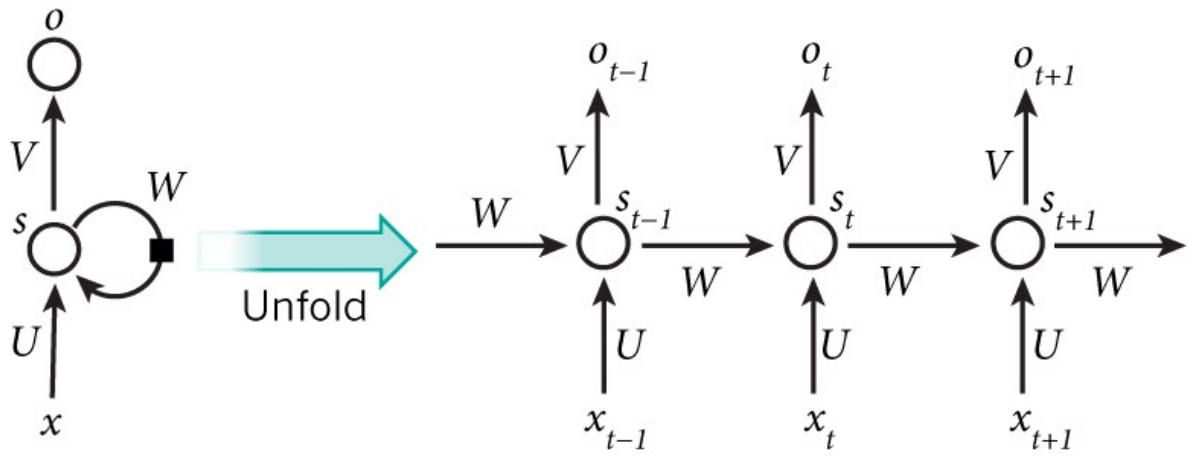
Recurrent Neural Networks (RNN)

Recurrent Neural Networks or RNN as they are called in short, are a very important variant of neural networks heavily used in Natural Language Processing. In a general neural network, an input is processed through a number of layers and an output is produced, with an assumption that two successive inputs are independent of each other.

This assumption is however not true in a number of real-life scenarios. For instance, if one wants to predict the price of a stock at a given time or wants to predict the next word in a sequence it is imperative that dependence on previous observations is considered.

RNNs are called *recurrent* because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far. In theory, RNNs can make use of information in arbitrarily long sequences, but in practice, they are limited to looking back only a few steps. [7]

Architecture wise, an RNN looks like this. One can imagine it as a multilayer neural network with each layer representing the observations at a certain time t.



RNN has shown to be hugely successful in natural language processing especially with their variant LSTM, which are able to look back longer than RNN. If you are interested in understanding LSTM, I would certainly encourage you to visit

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

In this article I have tried to cover neural networks from a theoretical standpoint, starting from the most basic structure, a neuron and covering up to the most popular versions of neural networks. The aim of this write up was to make readers understand how a neural network is built from scratch, which all fields it is used and what are its most successful variations.

I understand that there are many other popular versions which I will try to cover in subsequent posts. Please feel free to suggest a topic if you want it to be covered earlier.

http://timdettmers.com/2015/03/26/convolution-deep-learning/

Understanding Convolution in Deep Learning

2015-03-26 by Tim Dettmers 109 Comments

Convolution is probably the most important concept in deep learning right now. It was convolution and convolutional nets that catapulted deep learning to the forefront of almost any machine learning task there is. But what makes convolution so powerful? How does it work? In this blog post I will explain convolution and relate it to other concepts that will help you to understand convolution thoroughly.

There are already some blog post regarding convolution in deep learning, but I found all of them highly confusing with unnecessary mathematical details that do not further the understanding in any meaningful way. This blog post will also have many mathematical details, but I will approach them from a conceptual point of view where I represent the underlying mathematics with images everybody should be able to understand. The first part of this blog post is aimed at anybody who wants to understand the general concept of convolution and convolutional nets in deep learning. The second part of this blog post includes advanced concepts and is aimed to further and enhance the understanding of convolution for deep learning researchers and specialists.

What is convolution?

This whole blog post will build up to answer exactly this question, but it may be very helpful to first understand in which direction this is going, so what is convolution in rough terms?

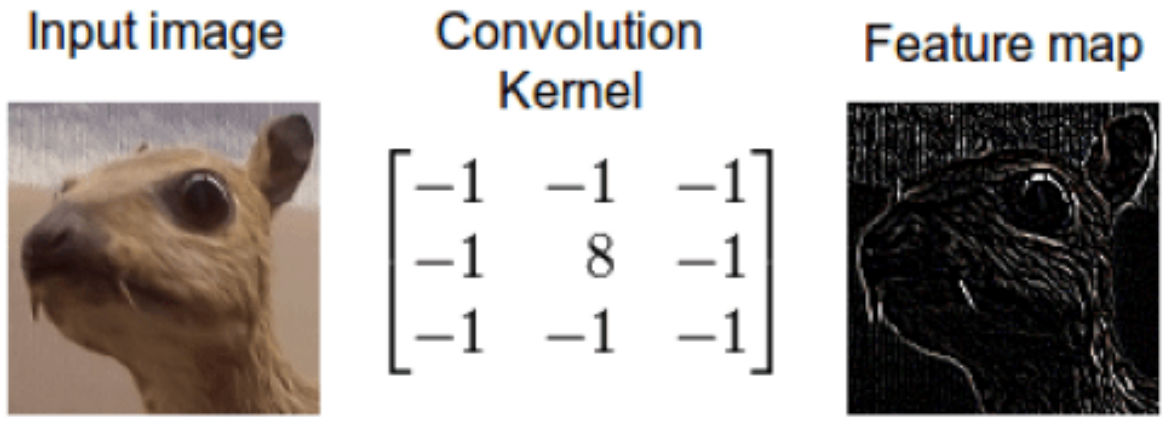
You can imagine convolution as the mixing of information. Imagine two buckets full of information which are poured into one single bucket and then mixed according to a specific rule. Each bucket of information has its own recipe, which describes how the information in one bucket mixes with the other. So convolution is an orderly procedure where two sources of information are intertwined.

Convolution can also be described mathematically, in fact, it is a mathematical operation like addition, multiplication or a derivative, and while this operation is complex in itself, it can be very useful to simplify even more complex equations. Convolutions are heavily used in physics and engineering to simplify such complex equations and in the second part — after a short mathematical development of convolution — we will relate and integrate ideas between these fields of science and deep learning to gain a deeper understanding of convolution. But for now we will look at convolution from a practical perspective.

How do we apply convolution to images?

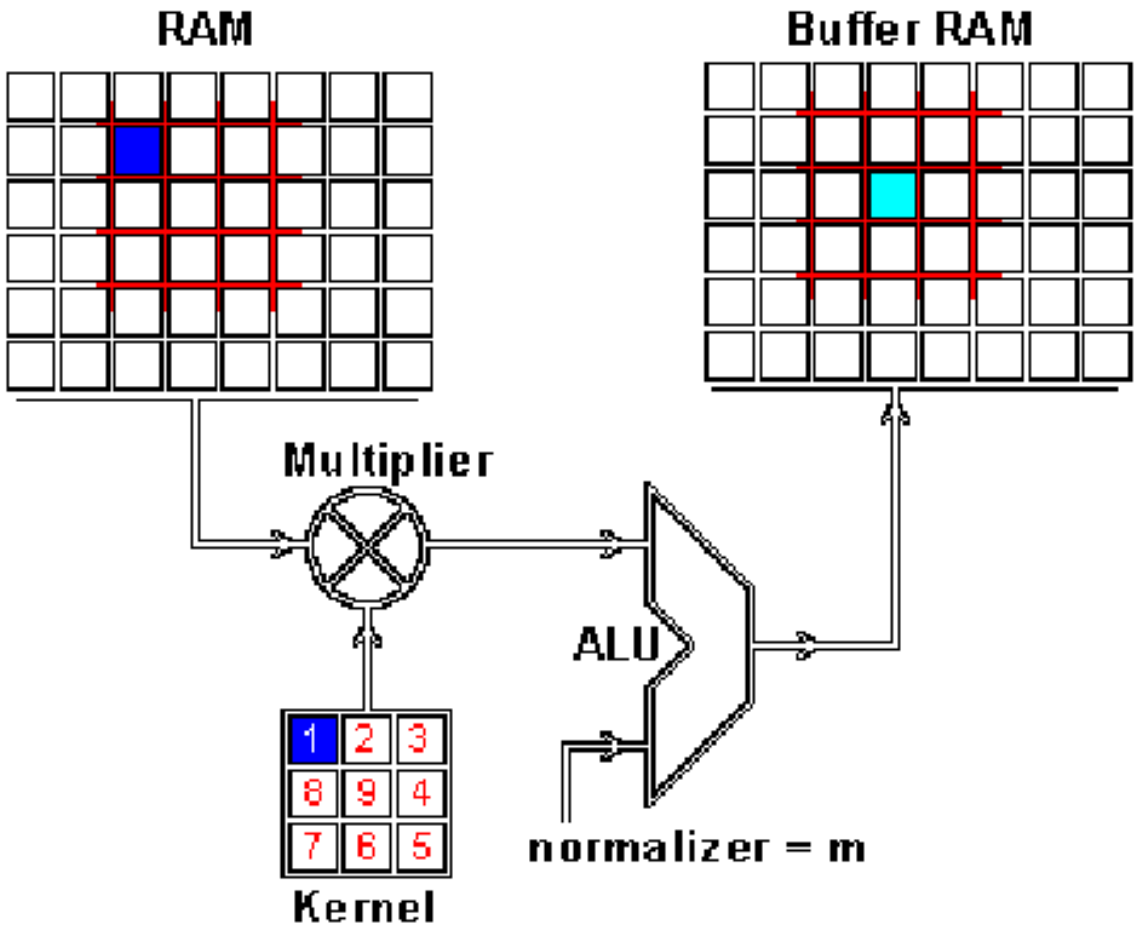
When we apply convolution to images, we apply it in two dimensions — that is the width and height of the image. We mix two buckets of information: The first bucket is the input image,

which has a total of three matrices of pixels — one matrix each for the red, blue and green color channels; a pixel consists of an integer value between 0 and 255 in each color channel. The second bucket is the convolution kernel, a single matrix of floating point numbers where the pattern and the size of the numbers can be thought of as a recipe for how to intertwine the input image with the kernel in the convolution operation. The output of the kernel is the altered image which is often called a feature map in deep learning. There will be one feature map for every color channel.



Convolution of an image with an edge detector convolution kernel. Sources: 1 2

We now perform the actual intertwining of these two pieces of information through convolution. One way to apply convolution is to take an image patch from the input image of the size of the kernel — here we have a 100×100 image, and a 3×3 kernel, so we would take 3×3 patches — and then do an element wise multiplication with the image patch and convolution kernel. The sum of this multiplication then results in *one* pixel of the feature map. After one pixel of the feature map has been computed, the center of the image patch extractor slides one pixel into another direction, and repeats this computation. The computation ends when all pixels of the feature map have been computed this way. This procedure is illustrated for one image patch in the following gif.



Convolution operation for one pixel of the resulting feature map: One image patch (red) of the original image (RAM) is multiplied by the kernel, and its sum is written to the feature map pixel (Buffer RAM). Gif by Glen Williamson who runs a website that features many technical gifs.

As you can see there is also a normalization procedure where the output value is normalized by the size of the kernel (9); this is to ensure that the total intensity of the picture and the feature map stays the same.

Why is convolution of images useful in machine learning?

There can be a lot of distracting information in images that is not relevant to what we are trying to achieve. A good example of this is a project I did together with Jannek Thomas in the Burda Bootcamp. The Burda Bootcamp is a rapid prototyping lab where students work in a hackathon-style environment to create technologically risky products in very short intervals. Together with my 9 colleagues, we created 11 products in 2 months. In one project I wanted to build a fashion image search with deep autoencoders: You upload an image of a fashion item and the autoencoder should find images that contain clothes with similar style.

Now if you want to differentiate between styles of clothes, the colors of the clothes will not be that useful for doing that; also minute details like emblems of the brand will be rather unimportant. What is most important is probably the shape of the clothes. Generally, the shape of a blouse is very different from the shape of a shirt, jacket, or trouser. So if we could filter the

unnecessary information out of images then our algorithm will not be distracted by the unnecessary details like color and branded emblems. We can achieve this easily by convoluting images with kernels.

My colleague Jannek Thomas preprocessed the data and applied a Sobel edge detector (similar to the kernel above) to filter everything out of the image except the outlines of the shape of an object — this is why the application of convolution is often called filtering, and the kernels are often called filters (a more exact definition of this filtering processes will follow below). The resulting feature map from the edge detector kernel will be very helpful if you want to differentiate between different types of clothes, because only relevant shape information remains.

Sobel filtered inputs to and results from the trained autoencoder: The top-left image is the search query and the other images are the results which have an autoencoder code that is most similar to the search query as measured by cosine similarity. You see that the autoencoder really just looks at the shape of the search query and not its color. However, you can also see that this procedure does not work well for images of people wearing clothes (5th column) and that it is sensitive to the shapes of clothes hangers (4th column).

We can take this a step further: There are dozens of different kernels which produce many different feature maps, e.g. which sharpen the image (more details), or which blur the image

(less details), and each feature map may help our algorithm to do better on its task (details, like 3 instead of 2 buttons on your jacket might be important).

Using this kind of procedure — taking inputs, transforming inputs and feeding the transformed inputs to an algorithm — is called feature engineering. Feature engineering is very difficult, and there are little resources which help you to learn this skill. In consequence, there are very few people which can apply feature engineering skillfully to a wide range of tasks. Feature engineering is — hands down — the most important skill to score well in Kaggle competitions. Feature engineering is so difficult because for each type of data and each type of problem, different features do well: Knowledge of feature engineering for image tasks will be quite useless for time series data; and even if we have two similar image tasks, it will not be easy to engineer good features because the objects in the images also determine what will work and what will not. It takes a lot of experience to get all of this right.

So feature engineering is very difficult and you have to start from scratch for each new task in order to do well. But when we look at images, might it be possible to automatically find the kernels which are most suitable for a task?

Enter convolutional nets

Convolutional nets do exactly this. Instead of having fixed numbers in our kernel, we assign parameters to these kernels which will be trained on the data. As we train our convolutional net, the kernel will get better and better at filtering a given image (or a given feature map) for relevant information. This process is automatic and is called feature learning. Feature learning automatically generalizes to each new task: We just need to simply train our network to find new filters which are relevant for the new task. This is what makes convolutional nets so powerful — no difficulties with feature engineering!

Usually we do not learn a single kernel in convolutional nets, instead we learn a hierarchy of multiple kernels at the same time. For example a 32x16x16 kernel applied to a 256×256 image would produce 32 feature maps of size 241×241 (this is the standard size, the size may vary from implementation to implementation; $\text{image size} - \text{kernel size} + 1$). So automatically we learn 32 new features that have relevant information for our task in them. These feature then provide the inputs for the next kernel which filters the inputs again. Once we learned our hierarchical features, we simply pass them to a fully connected, simple neural network that combines them in order to classify the input image into classes. That is nearly all that there is to know about convolutional nets at a conceptual level (pooling procedures are important too, but that would be another blog post).

Part II: Advanced concepts

We now have a very good intuition of what convolution is, and what is going on in convolutional nets, and why convolutional nets are so powerful. But we can dig deeper to understand what is really going on within a convolution operation. In doing so, we will see that the original interpretation of computing a convolution is rather cumbersome and we can develop more sophisticated interpretations which will help us to think about convolutions much more broadly so that we can apply them on many different data. To achieve this deeper understanding the first step is to understand the convolution theorem.

The convolution theorem

To develop the concept of convolution further, we make use of the convolution theorem, which relates convolution in the time/space domain — where convolution features an unwieldy integral or sum — to a mere element wise multiplication in the frequency/Fourier domain. This theorem is very powerful and is widely applied in many sciences. The convolution theorem is also one of the reasons why the fast Fourier transform (FFT) algorithm is thought by some to be one of the most important algorithms of the 20th century.

$$h(x) = f \otimes g = \int_{-\infty}^{\infty} f(x-u)g(u)\,du = \mathcal{F}^{-1}\left(\sqrt{2\pi}\mathcal{F}[f]\mathcal{F}[g]\right)$$

$$\text{feature map} = \text{input} \otimes \text{kernel} = \sum_{y=0}^{\text{columns}} \left(\sum_{x=0}^{\text{rows}} \text{input}(x-a, y-b)\,\text{kernel}(x,y) \right) = \mathcal{F}^{-1}\left(\sqrt{2\pi}\mathcal{F}[\text{input}]\mathcal{F}[\text{kernel}]\right)$$

The first equation is the one dimensional continuous convolution theorem of two general continuous functions; the second equation is the 2D discrete convolution theorem for discrete image data. Here $\otimes$ denotes a convolution operation, $\mathcal{F}$ denotes the Fourier transform, $\mathcal{F}^{-1}$ the inverse Fourier transform, and $\sqrt{2\pi}$ is a normalization constant. Note that "discrete" here means that our data consists of a countable number of variables (pixels); and 1D means that our variables can be laid out in one dimension in a meaningful way, e.g. time is one dimensional (one second after the other), images are two dimensional (pixels have rows and columns), videos are three dimensional (pixels have rows and columns, and images come one after another).

To get a better understanding what happens in the convolution theorem we will now look at the interpretation of Fourier transforms with respect to digital image processing.

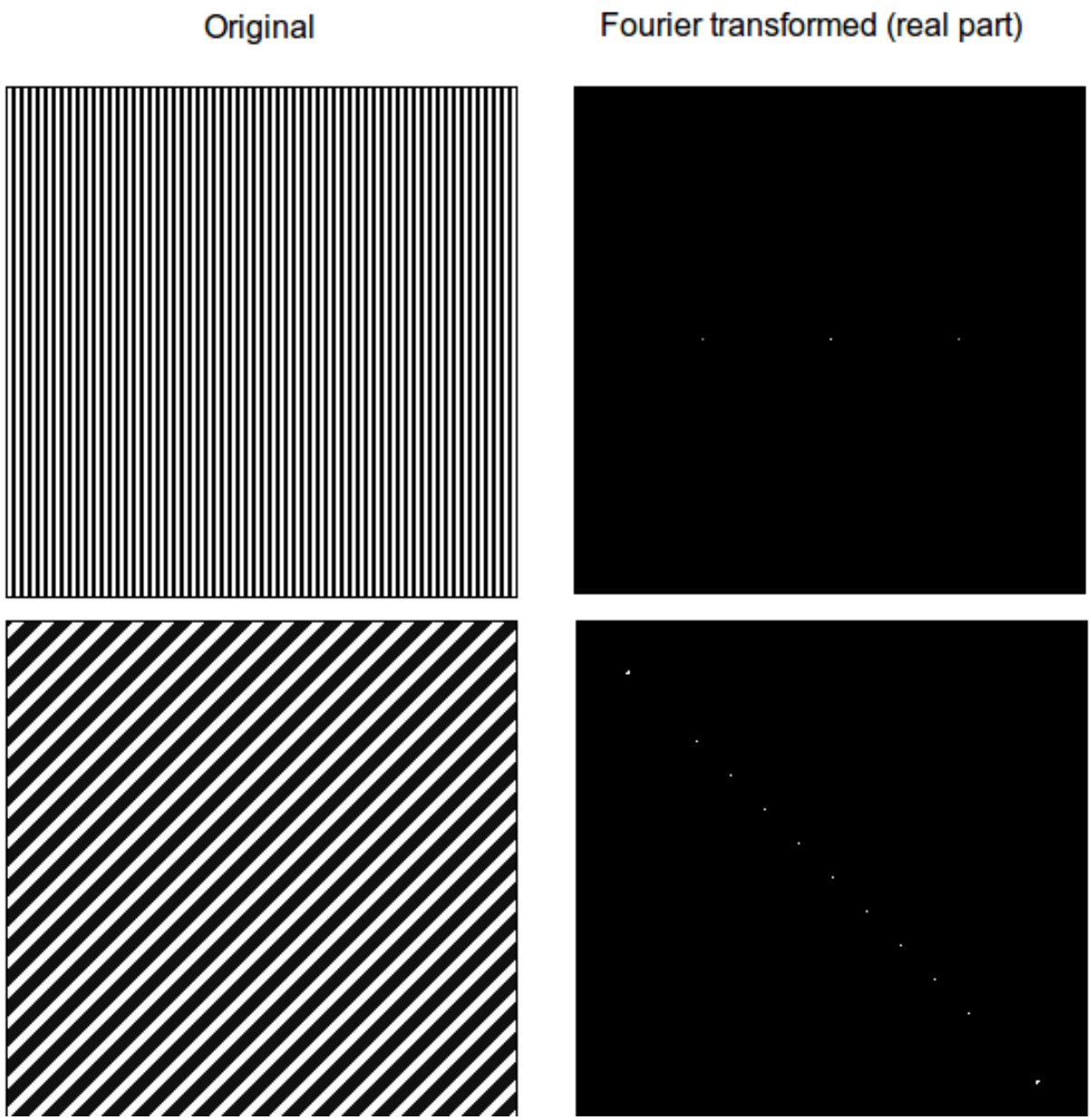
Fast Fourier transforms

The fast Fourier transform is an algorithm that transforms data from the space/time domain into the frequency or Fourier domain. The Fourier transform describes the original function in a sum of wave-like cosine and sine terms. It is important to note, that the Fourier transform is generally complex valued, which means that a real value is transformed into a complex value with a real and imaginary part. Usually the imaginary part is only important for certain operations and to transform the frequencies back into the space/time domain and will be largely ignored in this blog post. Below you can see a visualization how a signal (a function of information often with a time parameter, often periodic) is transformed by a Fourier transform.

Transformation of the time domain (red) into the frequency domain (blue). Source

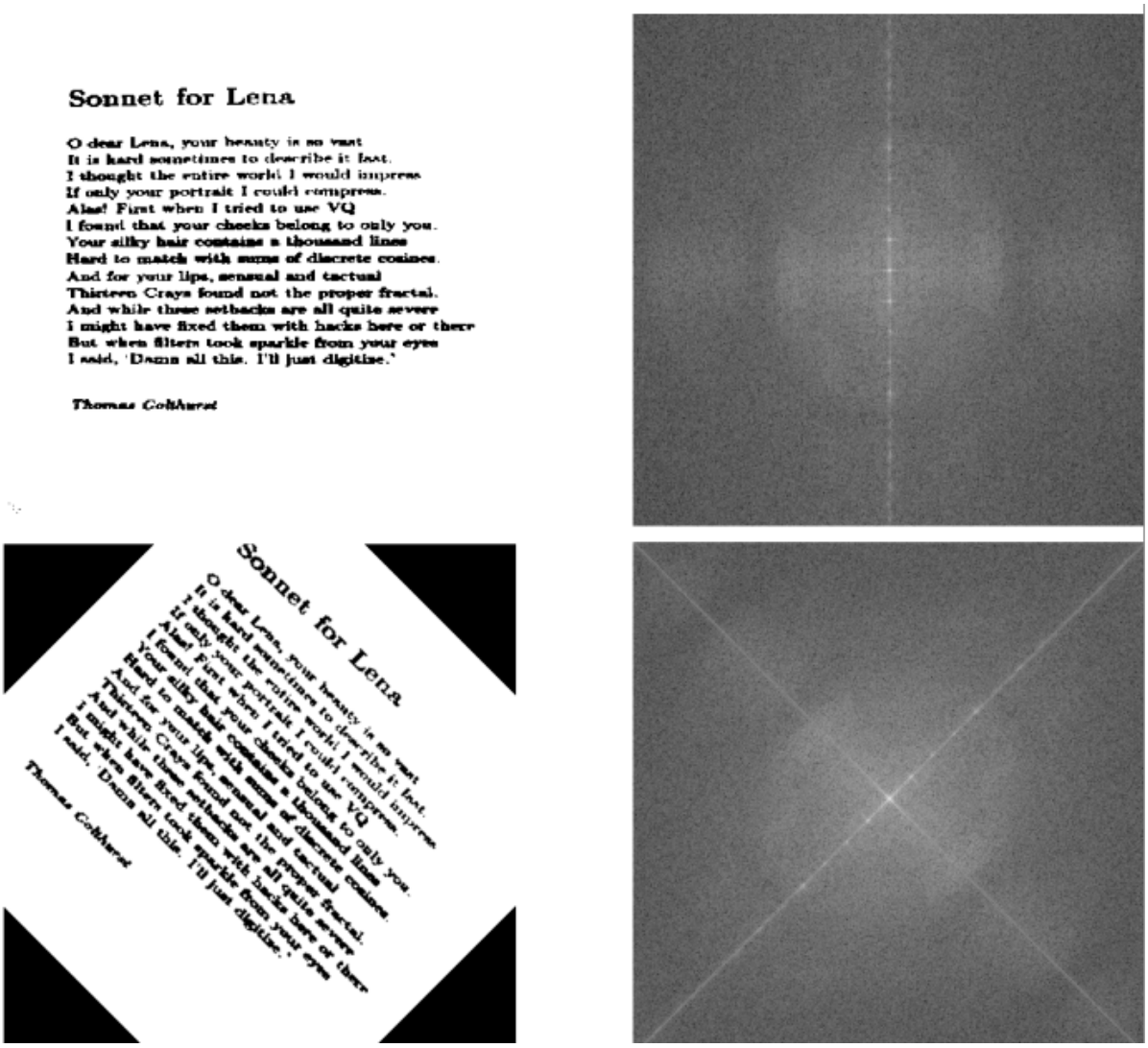
You may be unaware of this, but it might well be that you see Fourier transformed values on a daily basis: If the red signal is a song then the blue values might be the equalizer bars displayed by your mp3 player.

The Fourier domain for images



Images by Fisher & Koryllos (1998). Bob Fisher also runs an excellent website about Fourier transforms and image processing in general.

How can we imagine frequencies for images? Imagine a piece of paper with one of the two patterns from above on it. Now imagine a wave traveling from one edge of the paper to the other where the wave pierces through the paper at each stripe of a certain color and hovers over the other. Such waves pierce the black and white parts in specific intervals, for example, every two pixels — this represents the frequency. In the Fourier transform lower frequencies are closer to the center and higher frequencies are at the edges (the maximum frequency for an image is at the very edge). The location of Fourier transform values with high intensity (white in the images) are ordered according to the direction of the greatest change in intensity in the original image. This is very apparent from the next image and its log Fourier transforms (applying the log to the real values decreases the differences in pixel intensity in the image — we see information more easily this way).



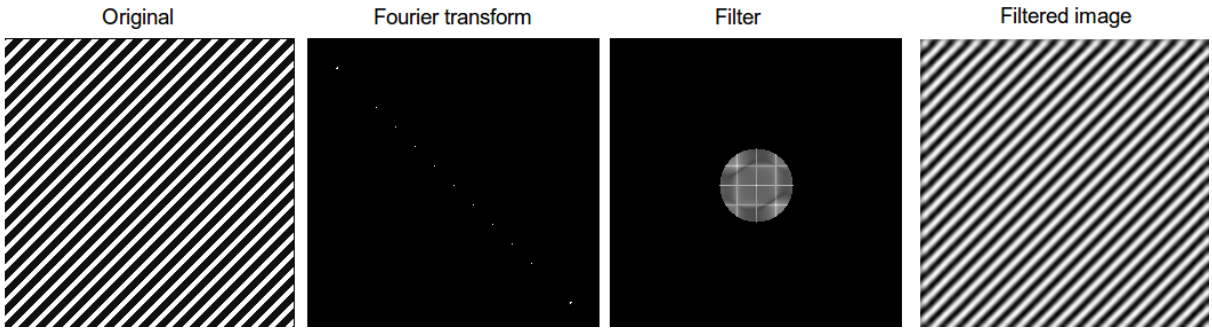
Images by Fisher & Koryllos (1998). Source

We immediately see that a Fourier transform contains a lot of information about the orientation of an object in an image. If an object is turned by, say, 37% degrees, it is difficult to tell that from the original pixel information, but very clear from the Fourier transformed values.

This is an important insight: Due to the convolution theorem, we can imagine that convolutional nets operate on images in the Fourier domain and from the images above we now know that images in that domain contain a lot of information about orientation. Thus convolutional nets should be better than traditional algorithms when it comes to rotated images and this is indeed the case (although convolutional nets are still very bad at this when we compare them to human vision).

Frequency filtering and convolution

The reason why the convolution operation is often described as a filtering operation, and why convolution kernels are often named filters will be apparent from the next example, which is very close to convolution.



Images by Fisher & Koryllos (1998). Source

If we transform the original image with a Fourier transform and then multiply it by a circle padded by zeros (zeros=black) in the Fourier domain, we filter out all high frequency values (they will be set to zero, due to the zero padded values). Note that the filtered image still has the same striped pattern, but its quality is much worse now — this is how jpeg compression works (although a different but similar transform is used), we transform the image, keep only certain frequencies and transform back to the spatial image domain; the compression ratio would be the size of the black area to the size of the circle in this example.

If we now imagine that the circle is a convolution kernel, then we have fully fledged convolution — just as in convolutional nets. There are still many tricks to speed up and stabilize the computation of convolutions with Fourier transforms, but this is the basic principle how it is done.

Now that we have established the meaning of the convolution theorem and Fourier transforms, we can now apply this understanding to different fields in science and enhance our interpretation of convolution in deep learning.

Insights from fluid mechanics

Fluid mechanics concerns itself with the creation of differential equation models for flows of fluids like air and water (air flows around an airplane; water flows around suspended parts of a bridge). Fourier transforms not only simplify convolution, but also differentiation, and this is why Fourier transforms are widely used in the field of fluid mechanics, or any field with differential equations for that matter. Sometimes the only way to find an analytic solution to a fluid flow problem is to simplify a partial differential equation with a Fourier transform. In this process we can sometimes rewrite the solution of such a partial differential equation in terms of a convolution of two functions which then allows for very easy interpretation of the solution. This is the case for the diffusion equation in one dimension, and for some two dimensional diffusion processes for functions in cylindrical or spherical polar coordinates.

Diffusion

You can mix two fluids (milk and coffee) by moving the fluid with an outside force (mixing with a spoon) — this is called convection and is usually very fast. But you could also wait and the two fluids would mix themselves on their own (if it is chemically possible) — this is called diffusion and is usually a very slow when compared to convection.

Imagine an aquarium that is split into two by a thin, removable barrier where one side of the aquarium is filled with salt water, and the other side with fresh water. If you now remove the

thin barrier carefully, the two fluids will mix together until the whole aquarium has the same concentration of salt everywhere. This process is more “violent” the greater the difference in saltiness between the fresh water and salt water.

Now imagine you have a square aquarium with 256×256 thin barriers that separate 256×256 cubes each with different salt concentration. If you remove the barrier now, there will be little mixing between two cubes with little difference in salt concentration, but rapid mixing between two cubes with very different salt concentrations. Now imagine that the 256×256 grid is an image, the cubes are pixels, and the salt concentration is the intensity of each pixel. Instead of diffusion of salt concentrations we now have diffusion of pixel information.

It turns out, this is exactly one part of the convolution for the diffusion equation solution: One part is simply the initial concentrations of a certain fluid in a certain area — or in image terms — the initial image with its initial pixel intensities. To complete the interpretation of convolution as a diffusion process we need to interpret the second part of the solution to the diffusion equation: The propagator.

Interpreting the propagator

The propagator is a probability density function, which denotes into which direction fluid particles diffuse over time. The problem here is that we do not have a probability function in deep learning, but a convolution kernel — how can we unify these concepts?

We can apply a normalization that turns the convolution kernel into a probability density function. This is just like computing the softmax for output values in a classification tasks. Here the softmax normalization for the edge detector kernel from the first example above.

$$Z = \text{softmax}\left[\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}\right] = \begin{pmatrix} 0.0001 & 0.0001 & 0.0001 \\ 0.0001 & 0.9992 & 0.0001 \\ 0.0001 & 0.0001 & 0.0001 \end{pmatrix}$$

Softmax of an edge detector: To calculate the softmax normalization, we taking each value [latex background="ffffff"]{x}[/latex] of the kernel and apply [latex background="ffffff"]{e^x}[/latex]. After that we divide by the sum of all [latex background="ffffff"]{e^x}[/latex]. Please note that this technique to calculate the softmax will be fine for most convolution kernels, but for more complex data the computation is a bit different to ensure numerical stability (floating point computation is inherently unstable for very large and very small values and you have to carefully navigate around troubles in this case).

Now we have a full interpretation of convolution on images in terms of diffusion. We can imagine the operation of convolution as a two part diffusion process: Firstly, there is strong diffusion where pixel intensities change (from black to white, or from yellow to blue, etc.) and secondly, the diffusion process in an area is regulated by the probability distribution of the convolution kernel. That means that each pixel in the kernel area, diffuses into another position within the kernel according to the kernel probability density.

For the edge detector above almost all information in the surrounding area will concentrate in a single space (this is unnatural for diffusion in fluids, but this interpretation is mathematically correct). For example all pixels that are under the 0.0001 values, will very likely flow into the center pixel and accumulate there. The final concentration will be largest where the largest differences between neighboring pixels are, because here the diffusion process is most marked. In turn, the greatest differences in neighboring pixels is there, where the edges between different objects are, so this explains why the kernel above is an edge detector.

So there we have it: Convolution as diffusion of information. We can apply this interpretation directly on other kernels. Sometimes we have to apply a softmax normalization for interpretation, but generally the numbers in itself say a lot about what will happen. Take the following kernel for example. Can you now interpret what that kernel is doing? Click here to find the solution (there is a link back to this position).

$$Z = \text{softmax}\left[\begin{pmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{pmatrix}\right] = \begin{pmatrix} 0.105 & 0.1125 & 0.105 \\ 0.1125 & 0.13 & 0.1125 \\ 0.105 & 0.1125 & 0.105 \end{pmatrix}$$

Wait, there is something fishy here

How come that we have deterministic behavior if we have a convolution kernel with probabilities? We have to interpret that single particles diffuse according to the probability distribution of the kernel, according to the propagator, don't we?

Yes, this is indeed true. However, if you take a tiny piece of fluid, say a tiny drop of water, you still have millions of water molecules in that tiny drop of water, and while a single molecule behaves stochastically according to the probability distribution of the propagator, a whole bunch of molecules have quasi deterministic behavior —this is an important interpretation from statistical mechanics and thus also for diffusion in fluid mechanics. We can interpret the probabilities of the propagator as the average distribution of information or pixel intensities; Thus our interpretation is correct from a viewpoint of fluid mechanics. However, there is also a valid stochastic interpretation for convolution.

Insights from quantum mechanics

The propagator is an important concept in quantum mechanics. In quantum mechanics a particle can be in a superposition where it has two or more properties which usually exclude themselves in our empirical world: For example, in quantum mechanics a particle can be at two places at the same time — that is a single object in two places.

However, when you measure the state of the particle — for example where the particle is right now — it will be either at one place or the other. In other terms, you destroy the superposition state by observation of the particle. The propagator then describes the probability distribution where you can expect the particle to be. So after measurement a particle might be — according to the probability distribution of the propagator — with 30% probability in place A and 70% probability in place B.

If we have entangled particles (spooky action at a distance), a few particles can hold hundreds or even millions of different states at the same time — this is the power promised by quantum computers.

So if we use this interpretation for deep learning, we can think that the pixels in an image are in a superposition state, so that in each image patch, each pixel is in 9 positions at the same time (if our kernel is 3×3). Once we apply the convolution we make a measurement and the superposition of each pixel collapses into a single position as described by the probability distribution of the convolution kernel, or in other words: For each pixel, we choose one pixel of the 9 pixels at random (with the probability of the kernel) and the resulting pixel is the average of all these pixels. For this interpretation to be true, this needs to be a true stochastic process, which means, the same image and the same kernel will generally yield different results. This interpretation does not relate one to one to convolution but it might give you ideas how to the apply convolution in stochastic ways or how to develop quantum algorithms for convolutional nets. A quantum algorithm would be able to calculate *all* possible combinations described by the kernel with *one* computation and in *linear* time/qubits with respect to the size of image and kernel.

Insights from probability theory

Convolution is closely related to cross-correlation. Cross-correlation is an operation which takes a small piece of information (a few seconds of a song) to filter a large piece of information (the whole song) for similarity (similar techniques are used on youtube to automatically tag videos for copyrights infringements).

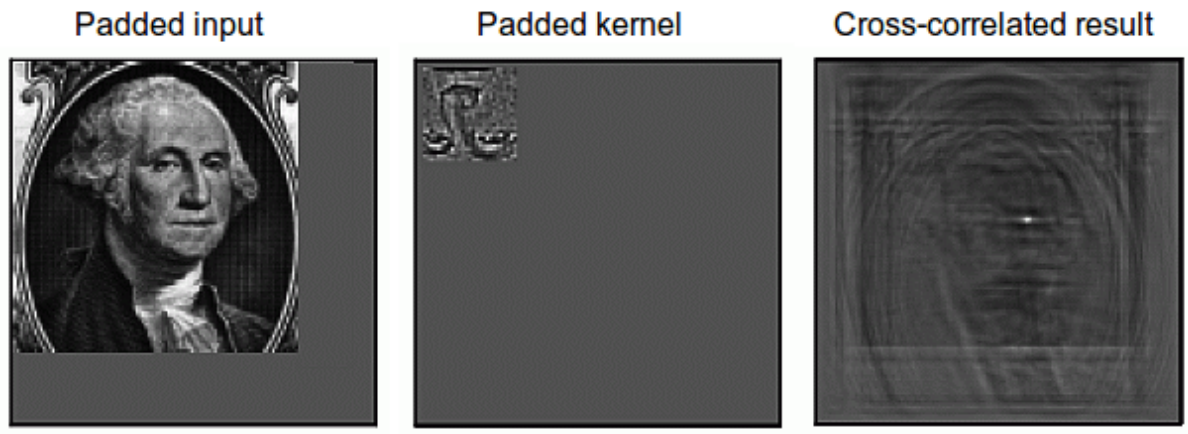
$$\text{convoluted } x = f \otimes g = \int_{-\infty}^{\infty} f(x-u)g(u)\,du = \mathcal{F}^{-1}\left(\sqrt{2\pi}\mathcal{F}[f]\mathcal{F}[g]\right)$$

$$\text{cross-correlated } x = f \star g = \int_{-\infty}^{\infty} f(x+u)g(u)^*\,du = \mathcal{F}^{-1}\left(\sqrt{2\pi}\mathcal{F}[f](\mathcal{F}[g])^*\right)$$

$$f(x) \star g(x) = f^*(-x) \otimes g(x)$$

Relation between cross-correlation and convolution: Here [latex background="ffffff"]{\star}[/latex] denotes cross correlation and [latex background="ffffff"]{f^*}[/latex] denotes the complex conjugate of [latex background="ffffff"]{f}[/latex].

While cross correlation seems unwieldy, there is a trick with which we can easily relate it to convolution in deep learning: For images we can simply turn the search image upside down to perform cross-correlation through convolution. When we perform convolution of an image of a person with an upside image of a face, then the result will be an image with one or multiple bright pixels at the location where the face was matched with the person.



Cross-correlation via convolution: The input and kernel are padded with zeros and the kernel is rotated by 180 degrees. The white spot marks the area with the strongest pixel-wise correlation between image and kernel. Note that the output image is in the spatial domain, the inverse Fourier transform was already applied. Images taken from Steven Smith's excellent free online book about digital signal processing.

This example also illustrates padding with zeros to stabilize the Fourier transform and this is required in many version of Fourier transforms. There are versions which require different padding schemes: Some implementation warp the kernel around itself and require only padding for the kernel, and yet other implementations perform divide-and-conquer steps and require no padding at all. I will not expand on this; the literature on Fourier transforms is vast and there are many tricks to be learned to make it run better — especially for images.

At lower levels, convolutional nets will not perform cross correlation, because we know that they perform edge detection in the very first convolutional layers. But in later layers, where more abstract features are generated, it is possible that a convolutional net learns to perform cross-correlation by convolution. It is imaginable that the bright pixels from the cross-correlation will be redirected to units which detect faces (the Google brain project has some units in its architecture which are dedicated to faces, cats etc.; maybe cross correlation plays a role here?).

Insights from statistics

What is the difference between statistical models and machine learning models? Statistical models often concentrate on very few variables which can be easily interpreted. Statistical models are built to answer questions: Is drug A better than drug B?

Machine learning models are about predictive performance: Drug A increases successful outcomes by 17.83% with respect to drug B for people with age X, but 22.34% for people with age Y.

Machine learning models are often much more powerful for prediction than statistical models, but they are not reliable. Statistical models are important to reach accurate and reliable conclusions: Even when drug A is 17.83% better than drug B, we do not know if this might be due to chance or not; we need statistical models to determine this.

Two important statistical models for time series data are the weighted moving average and the autoregressive models which can be combined into the ARIMA model (autoregressive

integrated moving average model). ARIMA models are rather weak when compared to models like long short-term recurrent neural networks, but ARIMA models are extremely robust when you have low dimensional data (1-5 dimensions). Although their interpretation is often effortful, ARIMA models are not a blackbox like deep learning algorithms and this is a great advantage if you need very reliable models.

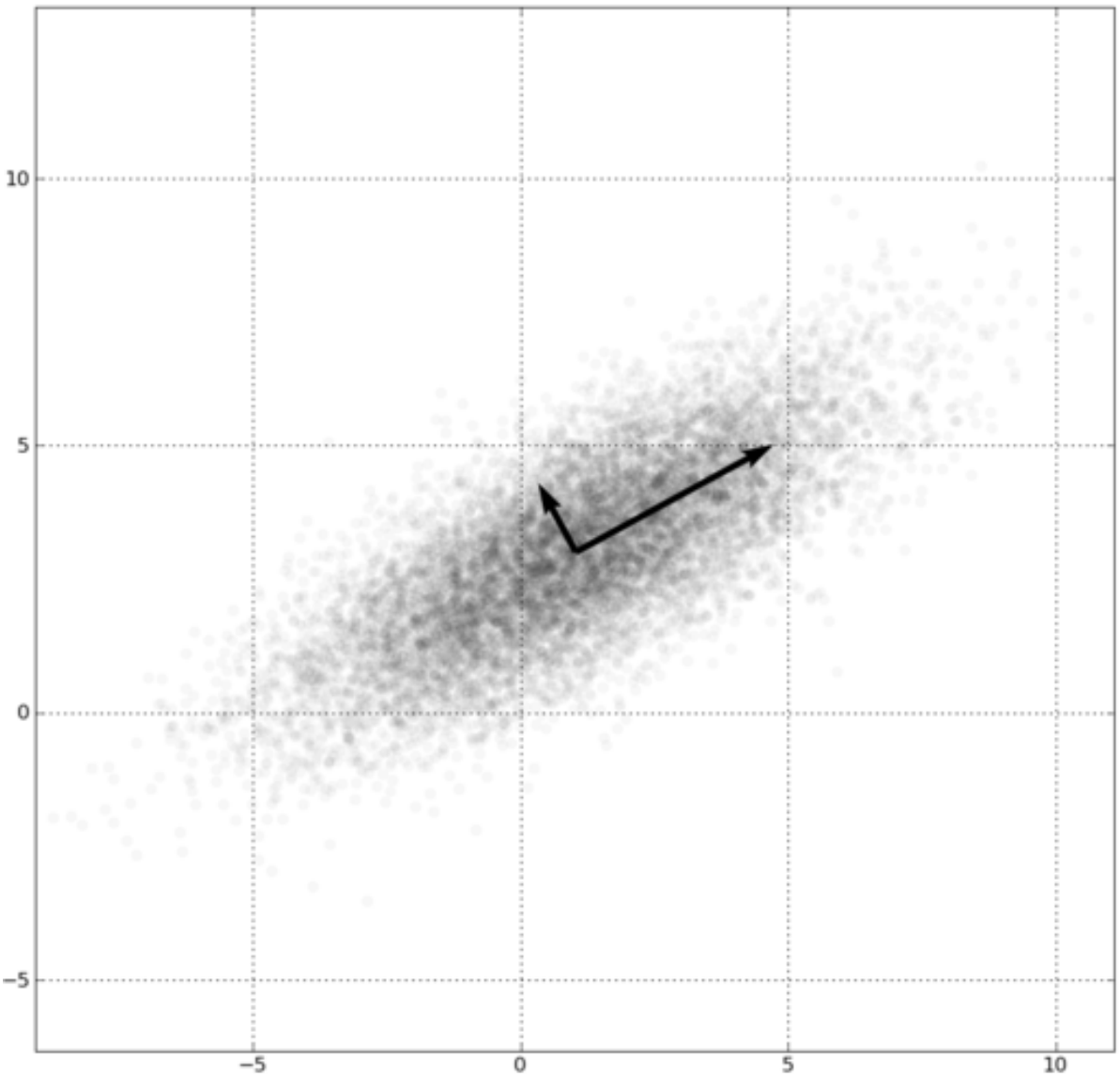
It turns out that we can rewrite these models as convolutions and thus we can show that convolutions in deep learning can be interpreted as functions which produce local ARIMA features which are then passed to the next layer. This idea however, does not overlap fully, and so we must be cautious and see when we really can apply this idea.

$$\text{autoregressed } x = C(\text{kernel}) + \text{white noise} \otimes \text{kernel}$$

$$\text{weighed moving averaged } x = \text{input} \otimes \text{kernel}$$

Here $C(\text{kernel})$ is a constant function which takes the kernel as parameter; white noise is data with mean zero, a standard deviation of one, and each variable is uncorrelated with respect to the other variables.

When we pre-process data we make it often very similar to white noise: We often center it around zero and set the variance/standard deviation to one. Creating uncorrelated variables is less often used because it is computationally intensive, however, conceptually it is straight forward: We reorient the axes along the eigenvectors of the data.



Decorrelation by reorientation along eigenvectors: The eigenvectors of this data are represented by the arrows. If we want to decorrelate the data, we reorient the axes to have the same direction as the eigenvectors. This technique is also used in PCA, where the dimensions with the least variance (shortest eigenvectors) are dropped after reorientation.

Now, if we take $C(\text{kernel})$ to be the bias, then we have an expression that is very similar to a convolution in deep learning. So the outputs from a convolutional layer can be interpreted as outputs from an autoregressive model if we pre-process the data to be white noise.

The interpretation of the weighted moving average is simple: It is just standard convolution on some data (input) with a certain weight (kernel). This interpretation becomes clearer when we look at the Gaussian smoothing kernel at the end of the page. The Gaussian smoothing kernel can be interpreted as a weighted average of the pixels in each pixel's neighborhood, or in other words, the pixels are averaged in their neighborhood (pixels "blend in", edges are smoothed).

While a single kernel cannot create both, autoregressive and weighted moving average features, we usually have multiple kernels and in combination all these kernels might contain some

features which are like a weighted moving average model and some which are like an autoregressive model.

Conclusion

In this blog post we have seen what convolution is all about and why it is so powerful in deep learning. The interpretation of image patches is easy to understand and easy to compute but it has many conceptual limitations. We developed convolutions by Fourier transforms and saw that Fourier transforms contain a lot of information about orientation of an image. With the powerful convolution theorem we then developed an interpretation of convolution as the diffusion of information across pixels. We then extended the concept of the propagator in the view of quantum mechanics to receive a stochastic interpretation of the usually deterministic process. We showed that cross-correlation is very similar to convolution and that the performance of convolutional nets may depend on the correlation between feature maps which is induced through convolution. Finally, we finished with relating convolution to autoregressive and moving average models.

Personally, I found it very interesting to work on this blog post. I felt for long time that my undergraduate studies in mathematics and statistics were wasted somehow, because they were so unpractical (even though I study *applied* math). But later — like an emergent property — all these thoughts linked together and practically useful understanding emerged. I think this is a great example why one should be patient and carefully study all university courses — even if they seem useless at first.

---

https://brohrer.github.io/how_convolutional_neural_networks_work.html

How do Convolutional Neural Networks work?

Data Science and Robots Blog

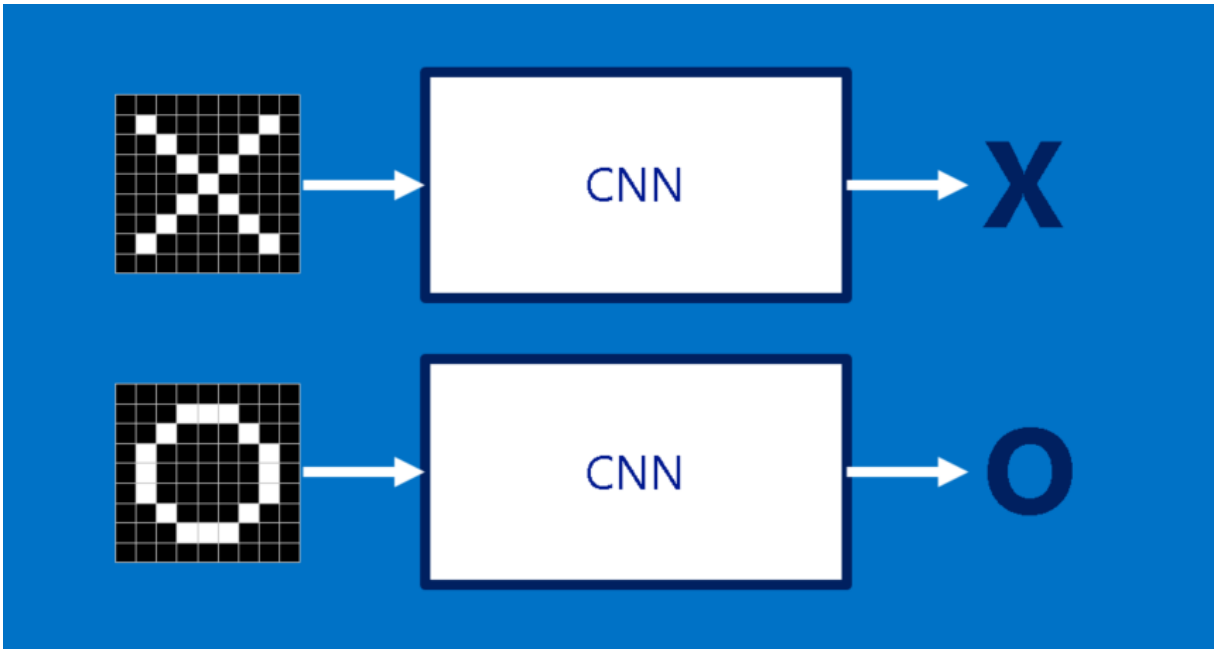
slides pdf [2MB] ppt [6MB]
in French by Charles Crouspeyre
in Japanese
in Simplified Mandarin by Jimmy Lin
in Traditional Mandarin by Jimmy Lin
in Persian by Elham Khanchebemehr
related presentation by Mohammad KHalooei

MATLAB and Caffe implementations for NVIDIA GPUs by Alexander Hanuschkin

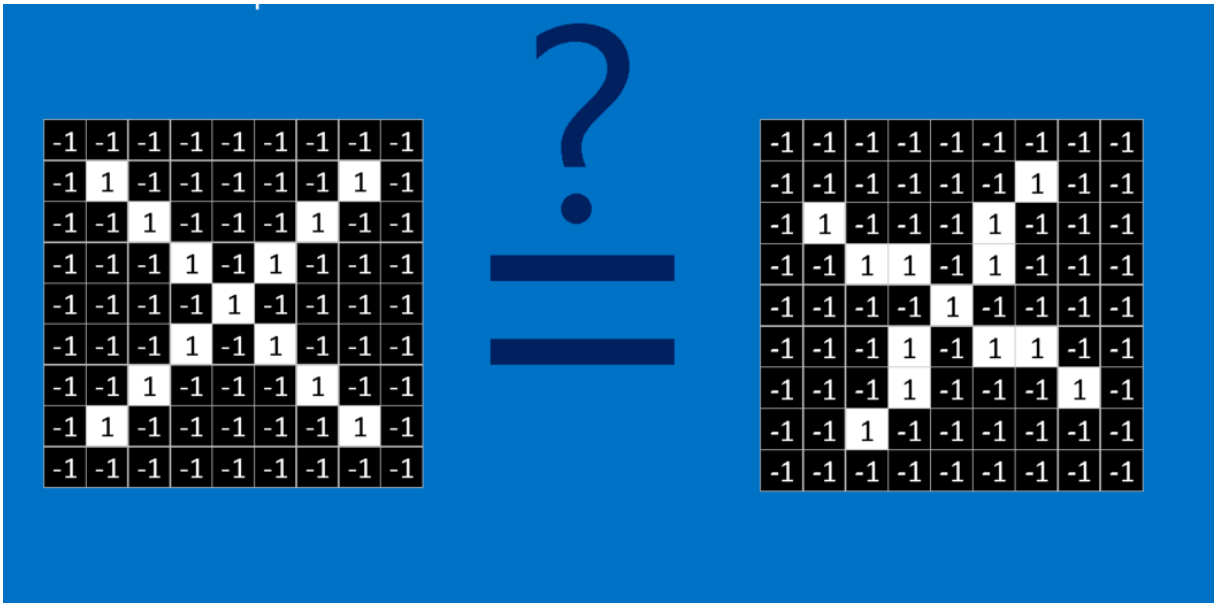
Nine times out of ten, when you hear about deep learning breaking a new technological barrier, Convolutional Neural Networks are involved. Also called CNNs or ConvNets, these are the workhorse of the deep neural network field. They have learned to sort images into categories even better than humans in some cases. If there's one method out there that justifies the hype, it is CNNs.

What's especially cool about them is that they are easy to understand, at least when you break them down into their basic parts. I'll walk you through it. There's a video that talks through these images in greater detail. If at any point you get a bit lost, just click on an image and you'll jump to that part of the video.

X's and O's

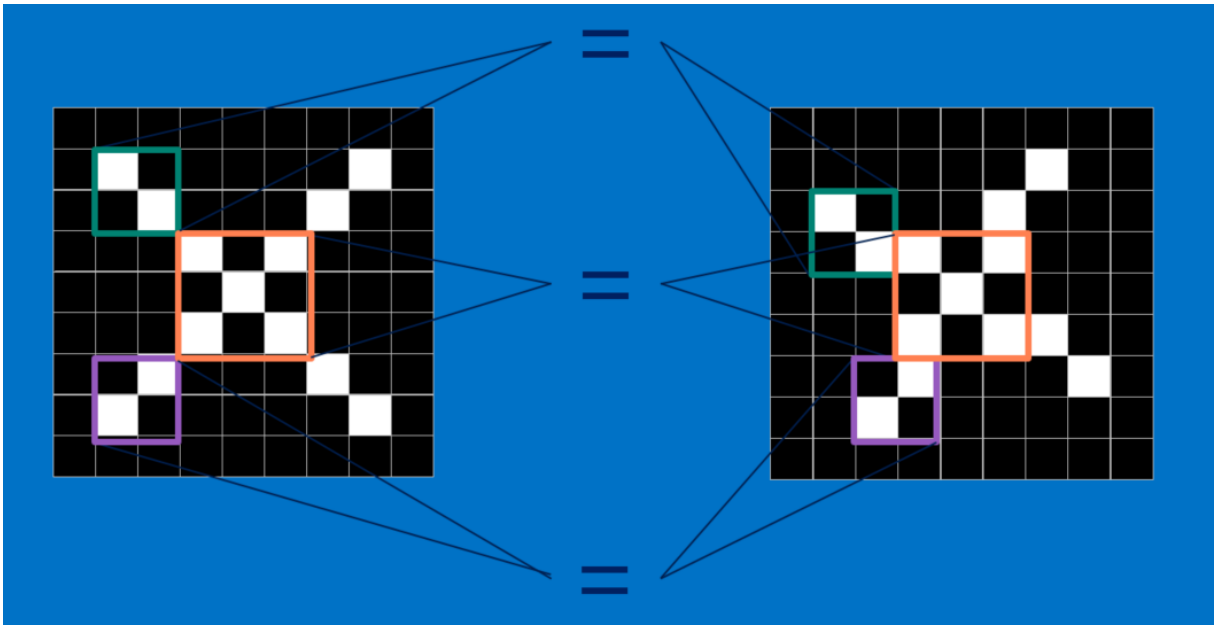


To help guide our walk through a Convolutional Neural Network, we'll stick with a very simplified example: determining whether an image is of an X or an O. This example is just rich enough to illustrate the principles behind CNNs, but still simple enough to avoid getting bogged down in non-essential details. Our CNN has one job. Each time we hand it a picture, it has to decide whether it has an X or an O. It assumes there is always one or the other.

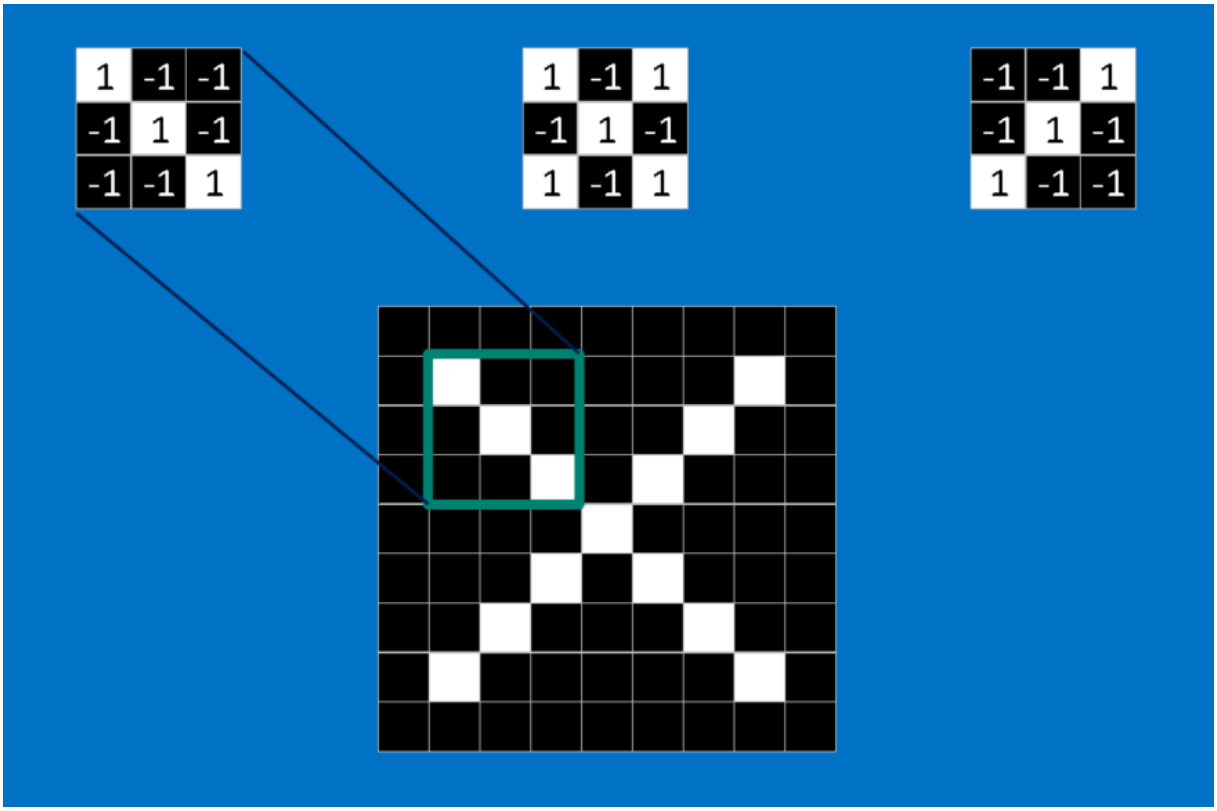


A naïve approach to solving this problem is to save an image of an X and an O and compare every new image to our exemplars to see which is the better match. What makes this task tricky is that computers are extremely literal. To a computer, an image looks like a two-dimensional array of pixels (think giant checkerboard) with a number in each position. In our example a pixel value of 1 is white, and -1 is black. When comparing two images, if any pixel values don't match, then the images don't match, at least to the computer. Ideally, we would like to be able to see X's and O's even if they're shifted, shrunken, rotated or deformed. This is where CNNs come in.

Features

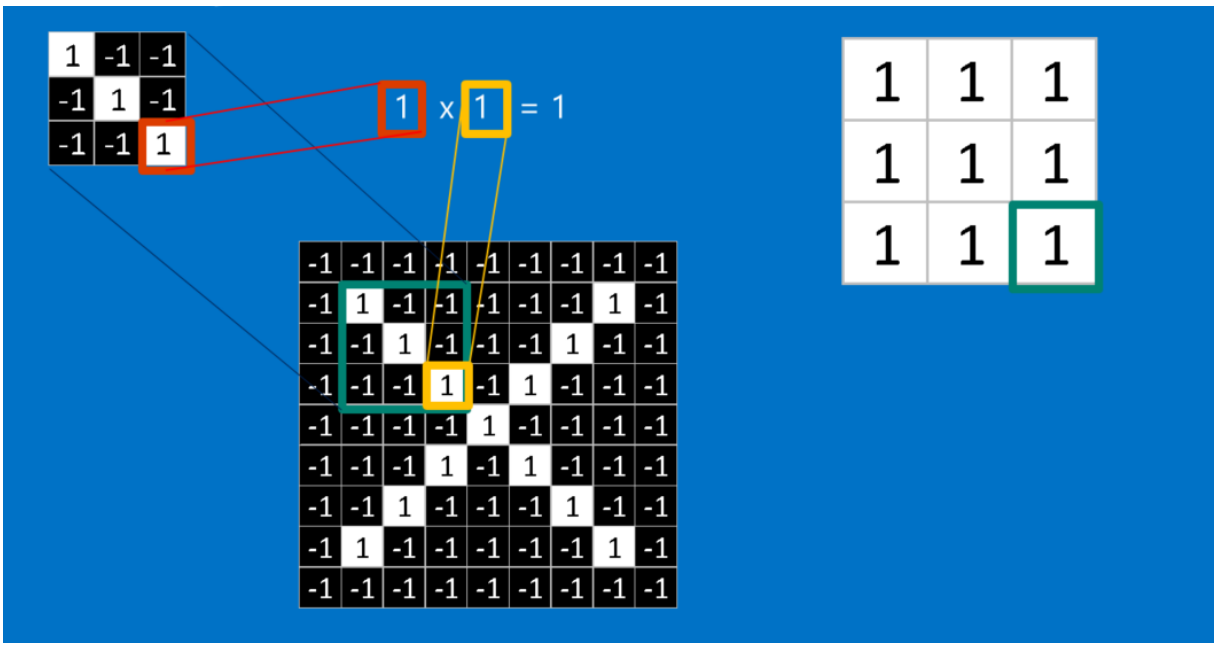


CNNs compare images piece by piece. The pieces that it looks for are called features. By finding rough feature matches in roughly the same positions in two images, CNNs get a lot better at seeing similarity than whole-image matching schemes.



Each feature is like a mini-image—a small two-dimensional array of values. Features match common aspects of the images. In the case of X images, features consisting of diagonal lines and a crossing capture all the important characteristics of most X's. These features will probably match up to the arms and center of any image of an X.

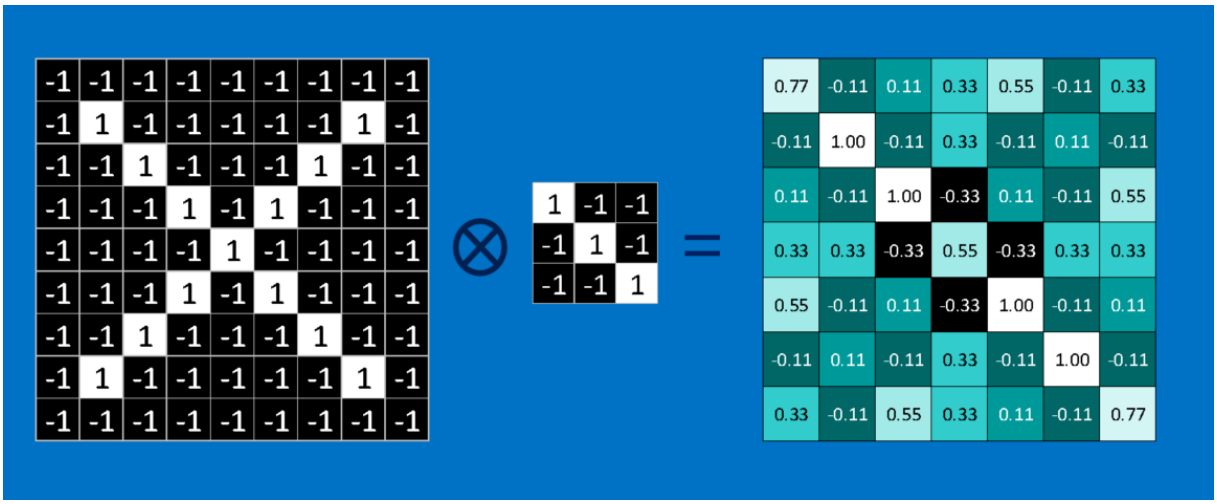
Convolution



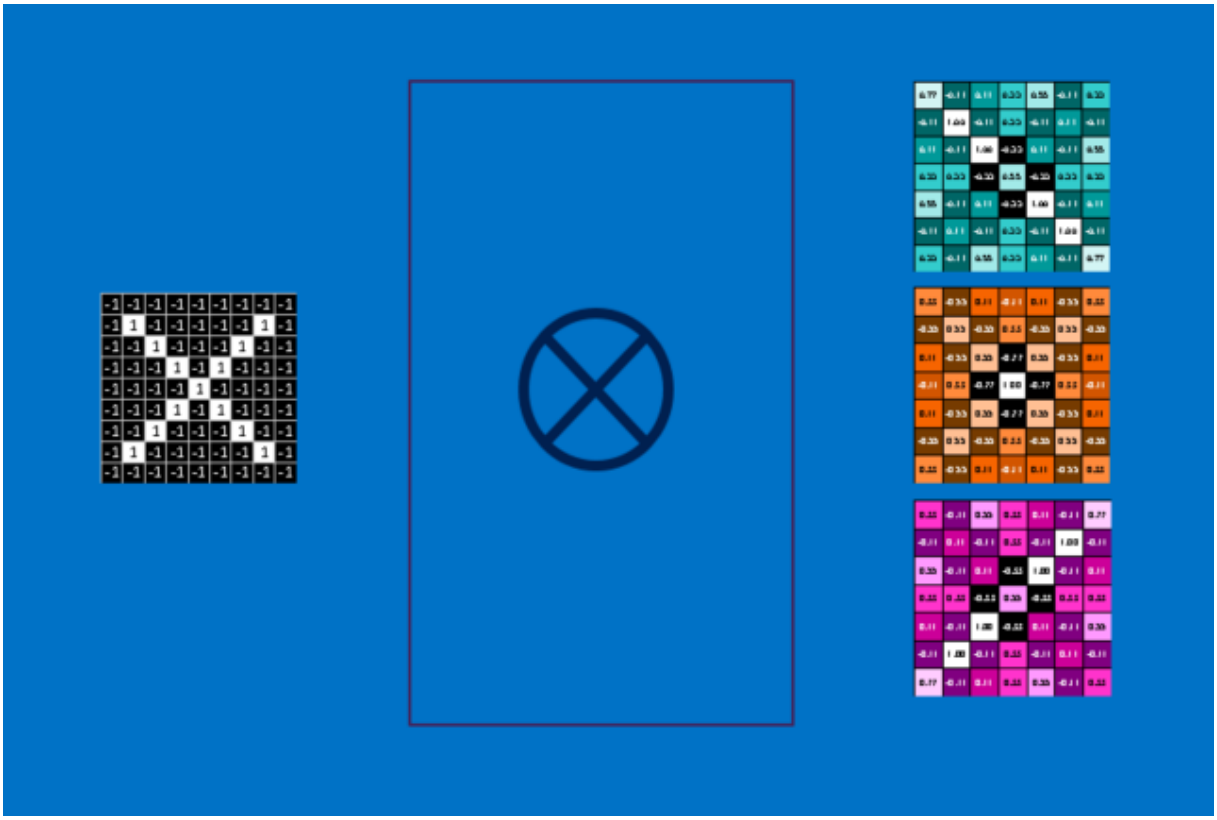
When presented with a new image, the CNN doesn't know exactly where these features will match so it tries them everywhere, in every possible position. In calculating the match to a

feature across the whole image, we make it a filter. The math we use to do this is called convolution, from which Convolutional Neural Networks take their name.

The math behind convolution is nothing that would make a sixth-grader uncomfortable. To calculate the match of a feature to a patch of the image, simply multiply each pixel in the feature by the value of the corresponding pixel in the image. Then add up the answers and divide by the total number of pixels in the feature. If both pixels are white (a value of 1) then 1 * 1 = 1. If both are black, then (-1) * (-1) = 1. Either way, every matching pixel results in a 1. Similarly, any mismatch is a -1. If all the pixels in a feature match, then adding them up and dividing by the total number of pixels gives a 1. Similarly, if none of the pixels in a feature match the image patch, then the answer is a -1.



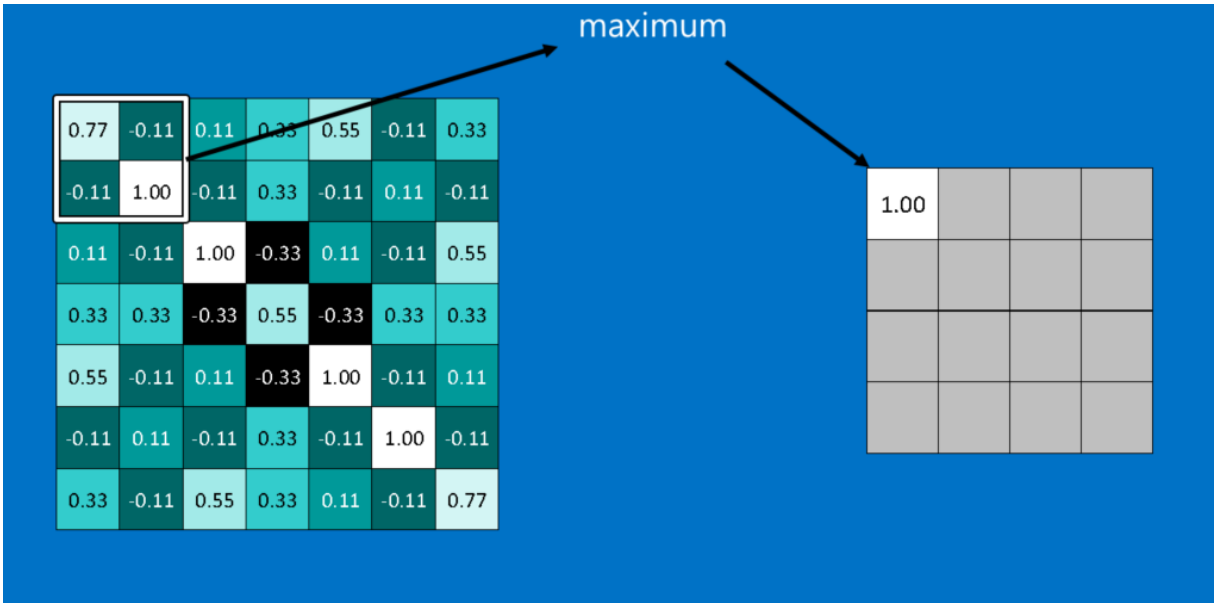
To complete our convolution, we repeat this process, lining up the feature with every possible image patch. We can take the answer from each convolution and make a new two-dimensional array from it, based on where in the image each patch is located. This map of matches is also a filtered version of our original image. It's a map of where in the image the feature is found. Values close to 1 show strong matches, values close to -1 show strong matches for the photographic negative of our feature, and values near zero show no match of any sort.



The next step is to repeat the convolution process in its entirety for each of the other features. The result is a set of filtered images, one for each of our filters. It's convenient to think of this whole collection of convolution operations as a single processing step. In CNNs this is referred to as a convolution layer, hinting at the fact that it will soon have other layers added to it.

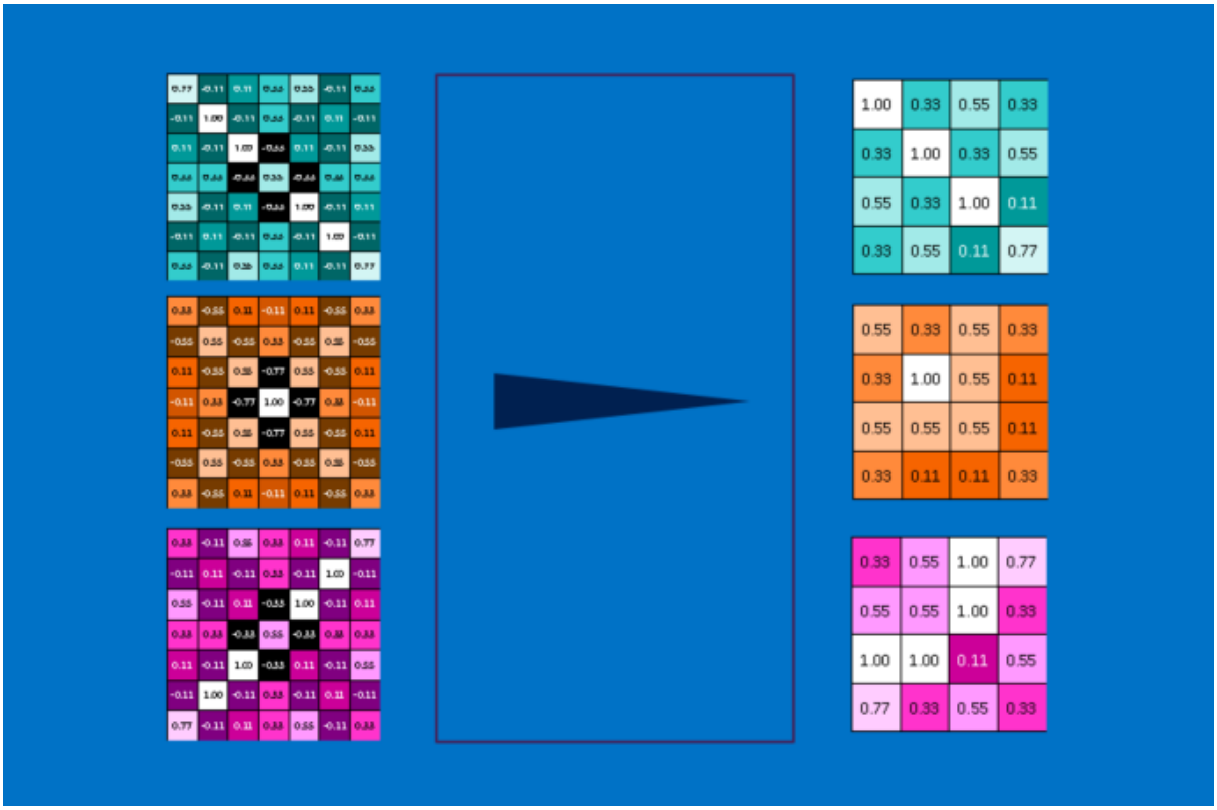
It's easy to see how CNNs get their reputation as computation hogs. Although we can sketch our CNN on the back of a napkin, the number of additions, multiplications and divisions can add up fast. In math speak, they scale linearly with the number of pixels in the image, with the number of pixels in each feature and with the number of features. With so many factors, it's easy to make this problem many millions of times larger without breaking a sweat. Small wonder that microchip manufacturers are now making specialized chips in an effort to keep up with the demands of CNNs.

Pooling



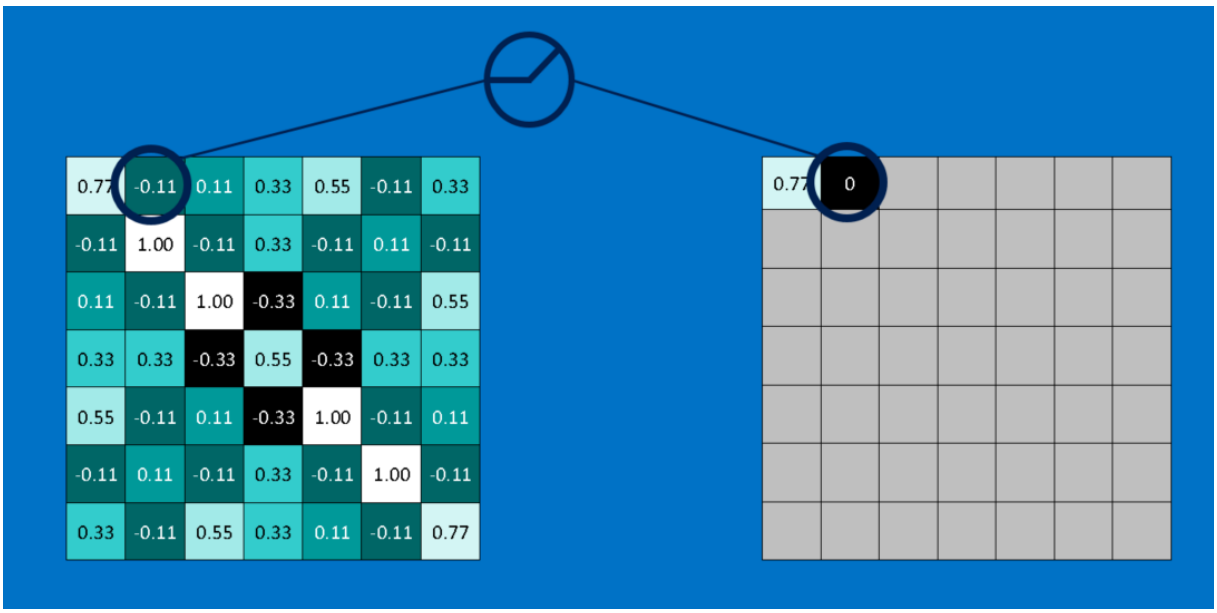
Another power tool that CNNs use is called pooling. Pooling is a way to take large images and shrink them down while preserving the most important information in them. The math behind pooling is second-grade level at most. It consists of stepping a small window across an image and taking the maximum value from the window at each step. In practice, a window 2 or 3 pixels on a side and steps of 2 pixels work well.

After pooling, an image has about a quarter as many pixels as it started with. Because it keeps the maximum value from each window, it preserves the best fits of each feature within the window. This means that it doesn't care so much exactly where the feature fit as long as it fit somewhere within the window. The result of this is that CNNs can find whether a feature is in an image without worrying about where it is. This helps solve the problem of computers being hyper-literal.



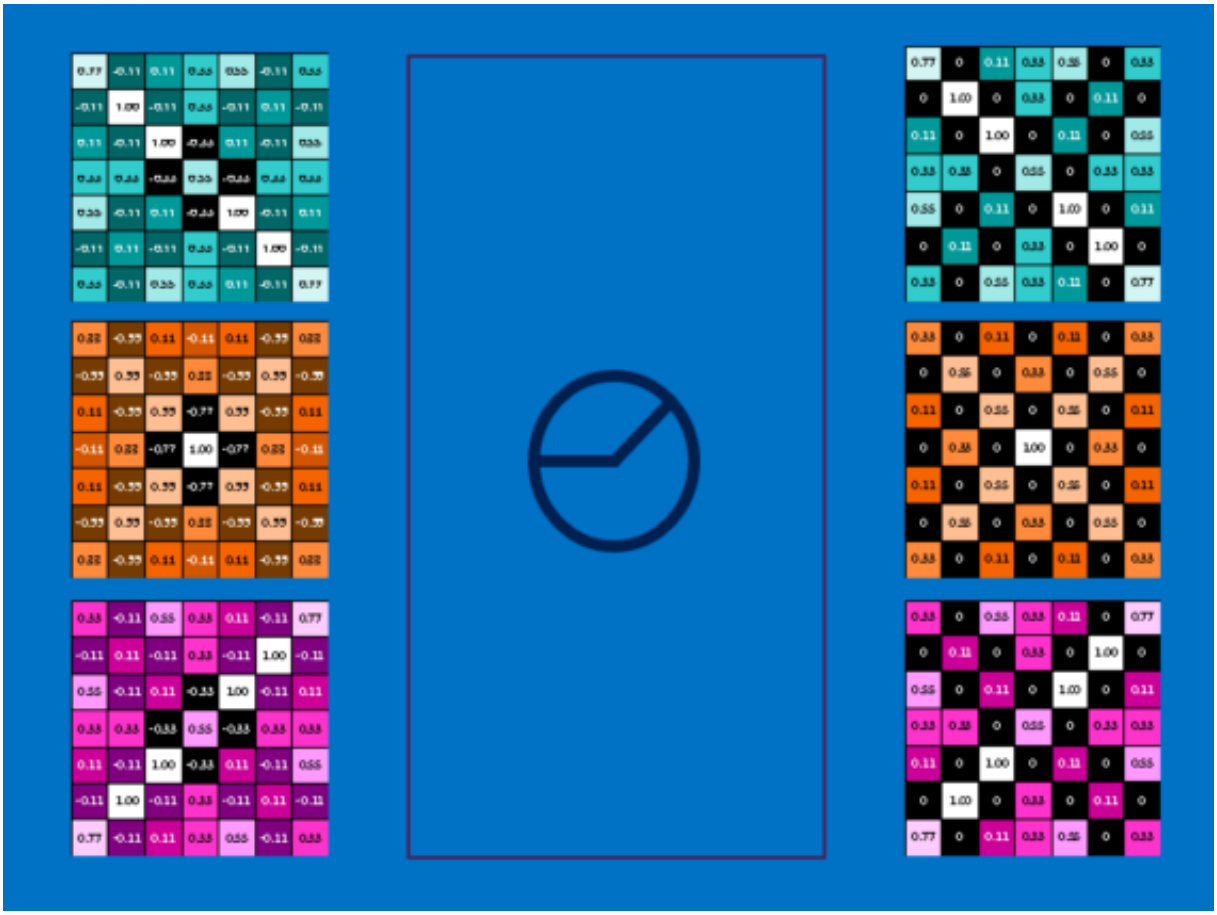
A pooling layer is just the operation of performing pooling on an image or a collection of images. The output will have the same number of images, but they will each have fewer pixels. This is also helpful in managing the computational load. Taking an 8 megapixel image down to a 2 megapixel image makes life a lot easier for everything downstream.

Rectified Linear Units



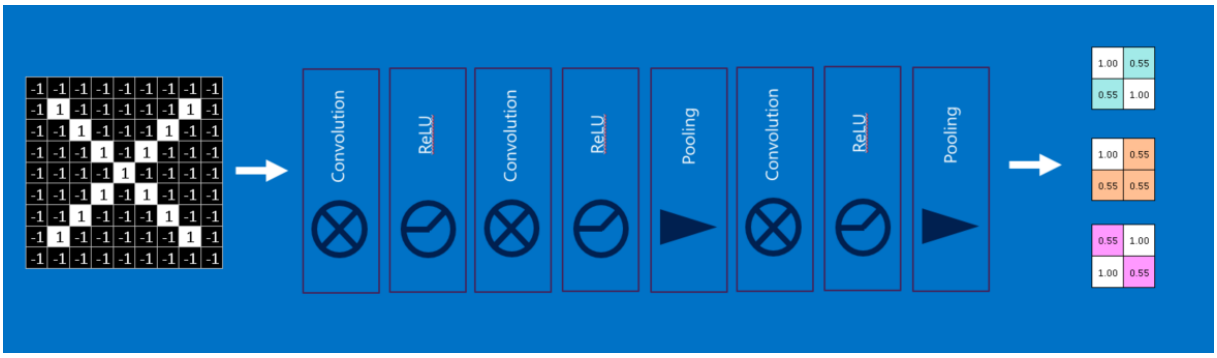
A small but important player in this process is the Rectified Linear Unit or ReLU. It's math is also very simple—wherever a negative number occurs, swap it out for a 0. This helps the CNN stay mathematically healthy by keeping learned values from getting stuck near 0 or

blowing up toward infinity. It's the axle grease of CNNs—not particularly glamorous, but without it they don't get very far.



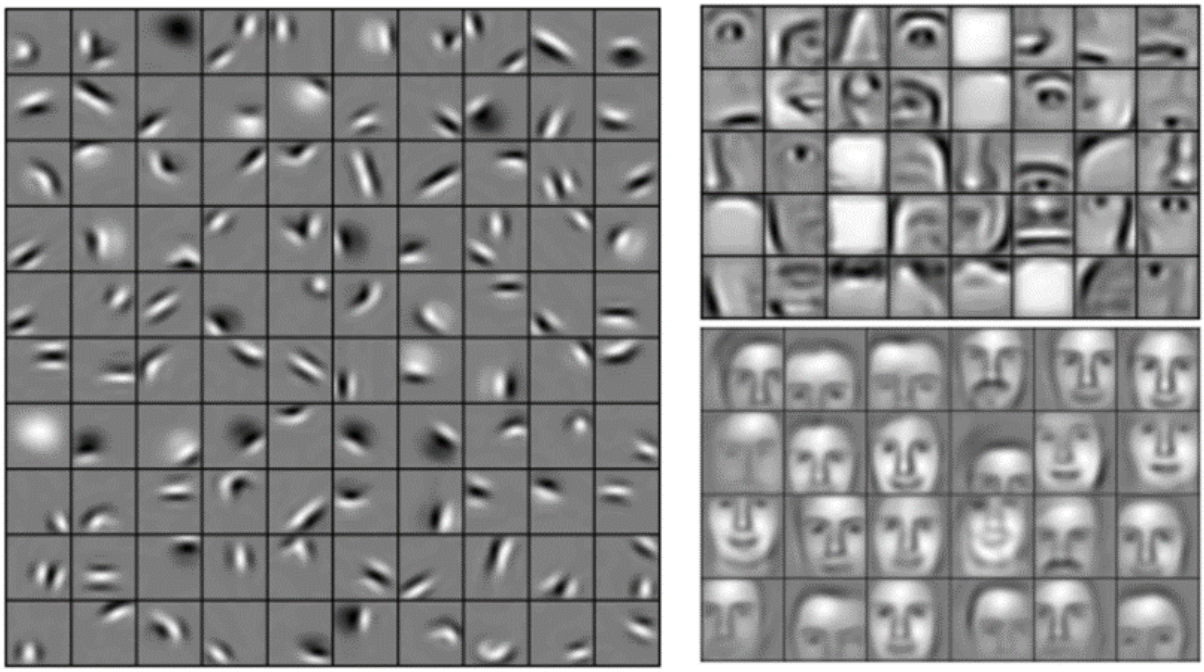
The output of a ReLU layer is the same size as whatever is put into it, just with all the negative values removed.

Deep learning

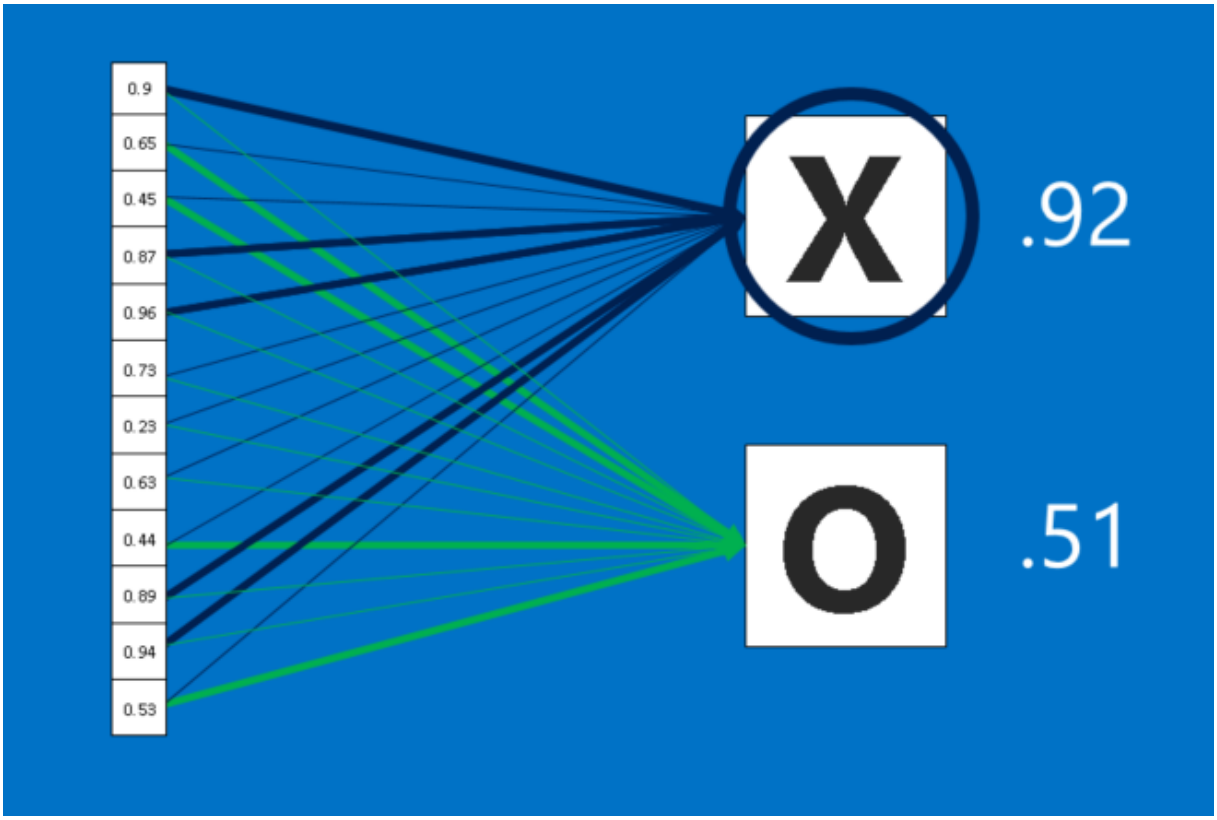


You've probably noticed that the input to each layer (two-dimensional arrays) looks a lot like the output (two-dimensional arrays). Because of this, we can stack them like Lego bricks. Raw images get filtered, rectified and pooled to create a set of shrunken, feature-filtered images. These can be filtered and shrunken again and again. Each time, the features become larger and more complex, and the images become more compact. This lets lower layers represent simple aspects of the image, such as edges and bright spots. Higher layers can

represent increasingly sophisticated aspects of the image, such as shapes and patterns. These tend to be readily recognizable. For instance, in a CNN trained on human faces, the highest layers represent patterns that are clearly face-like.



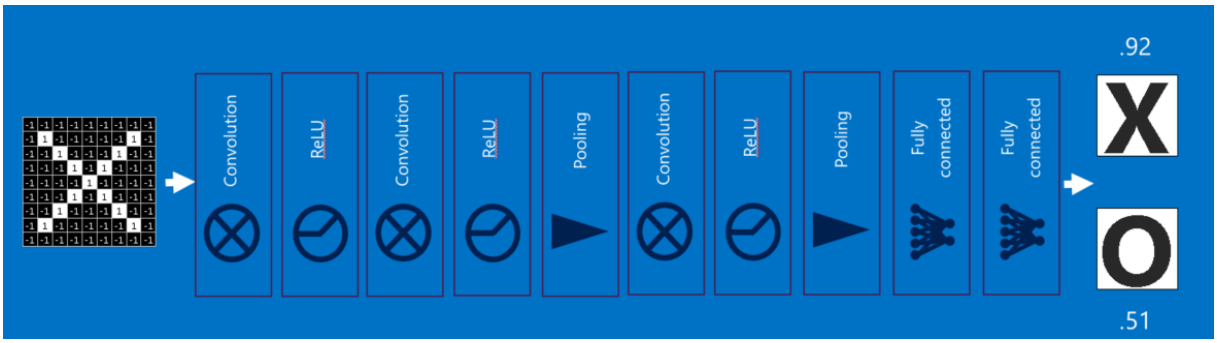
Fully connected layers



CNNs have one more arrow in their quiver. Fully connected layers take the high-level filtered images and translate them into votes. In our case, we only have to decide between two

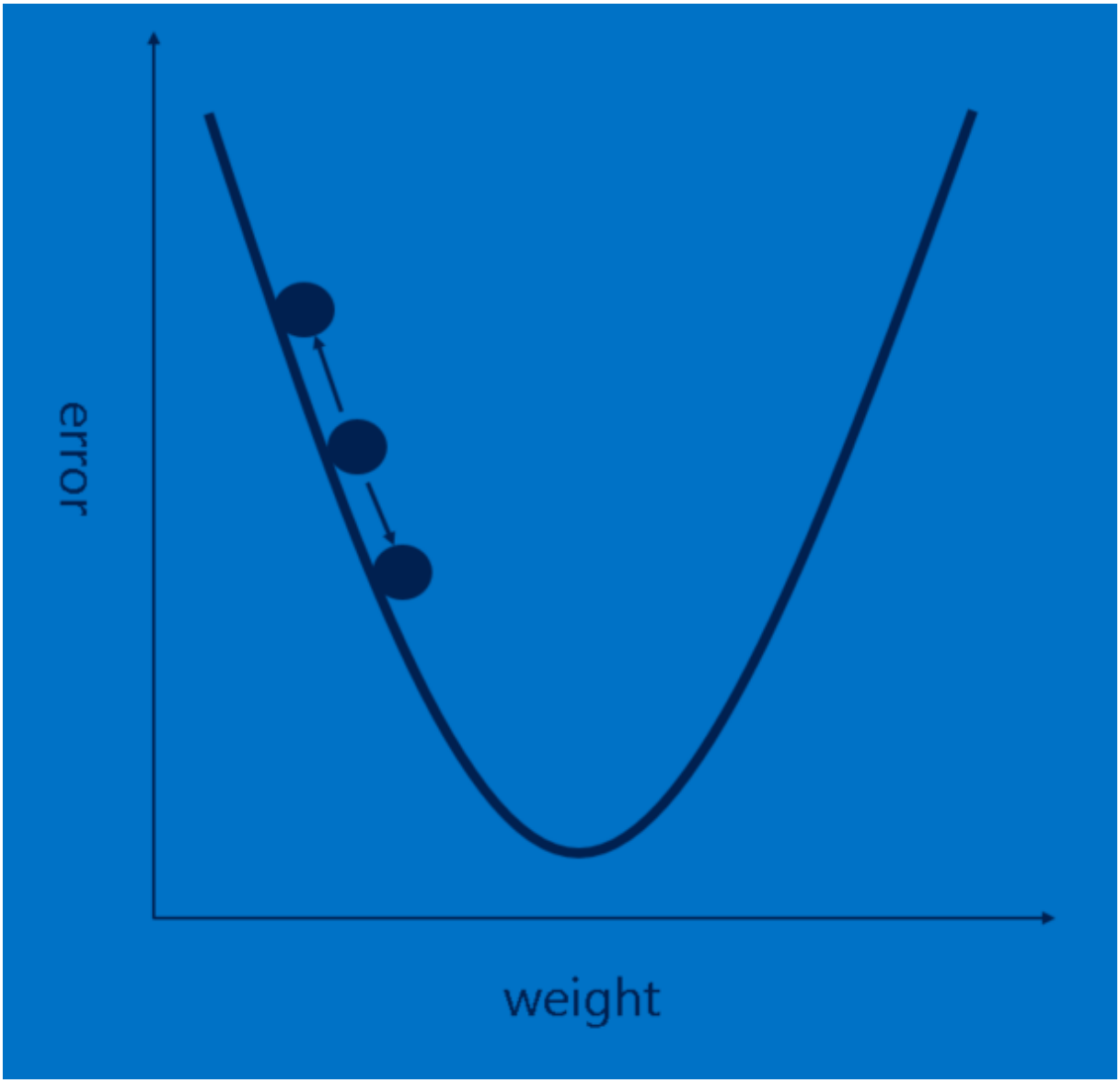
categories, X and O. Fully connected layers are the primary building block of traditional neural networks. Instead of treating inputs as a two-dimensional array, they are treated as a single list and all treated identically. Every value gets its own vote on whether the current image is an X or and O. However, the process isn't entirely democratic. Some values are much better than others at knowing when the image is an X, and some are particularly good at knowing when the image is an O. These get larger votes than the others. These votes are expressed as weights, or connection strengths, between each value and each category.

When a new image is presented to the CNN, it percolates through the lower layers until it reaches the fully connected layer at the end. Then an election is held. The answer with the most votes wins and is declared the category of the input.



Fully connected layers, like the rest, can be stacked because their outputs (a list of votes) look a whole lot like their inputs (a list of values). In practice, several fully connected layers are often stacked together, with each intermediate layer voting on phantom "hidden" categories. In effect, each additional layer lets the network learn ever more sophisticated combinations of features that help it make better decisions.

Backpropagation



Our story is filling in nicely, but it still has a huge hole—Where do features come from? and How do we find the weights in our fully connected layers? If these all had to be chosen by hand, CNNs would be a good deal less popular than they are. Luckily, a bit of machine learning magic called backpropagation does this work for us.

To make use of backpropagation, we need a collection of images that we already know the answer for. This means that some patient soul flipped through thousands of images and assigned them a label of X or O. We use these with an untrained CNN, which means that every pixel of every feature and every weight in every fully connected layer is set to a random value. Then we start feeding images through it, one after other.

Each image the CNN processes results in a vote. The amount of wrongness in the vote, the error, tells us how good our features and weights are. The features and weights can then be adjusted to make the error less. Each value is adjusted a little higher and a little lower, and the new error computed each time. Whichever adjustment makes the error less is kept. After doing this for every feature pixel in every convolutional layer and every weight in every fully connected layer, the new weights give an answer that works slightly better for that image.

This is then repeated with each subsequent image in the set of labeled images. Quirks that occur in a single image are quickly forgotten, but patterns that occur in lots of images get baked into the features and connection weights. If you have enough labeled images, these values stabilize to a set that works pretty well across a wide variety of cases.

As is probably apparent, backpropagation is another expensive computing step, and another motivator for specialized computing hardware.

Hyperparameters

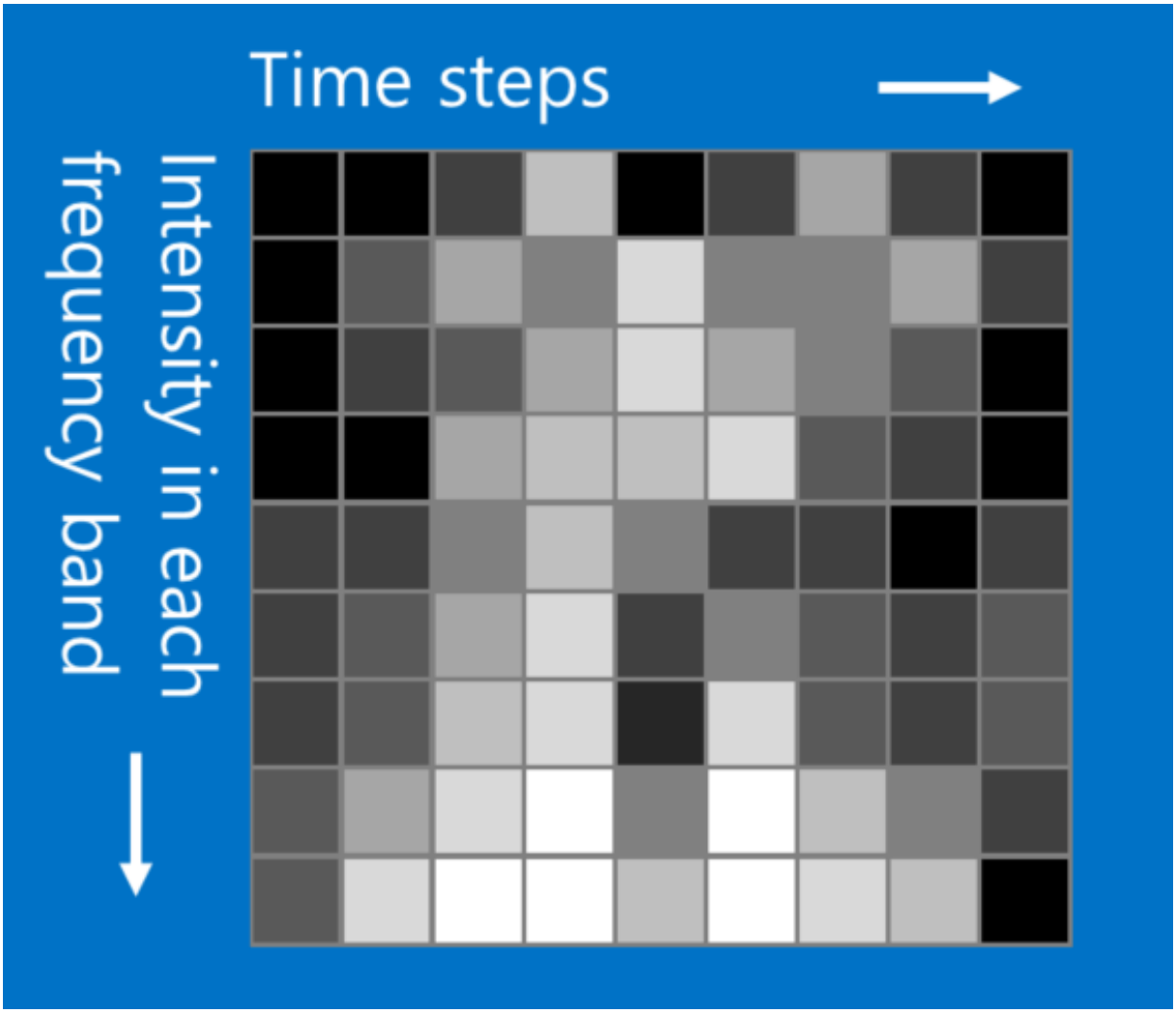
Unfortunately, not every aspect of CNNs can be learned in so straightforward a manner. There is still a long list of decisions that a CNN designer must make.

- For each convolution layer, How many features? How many pixels in each feature?
- For each pooling layer, What window size? What stride?
- For each extra fully connected layer, How many hidden neurons?

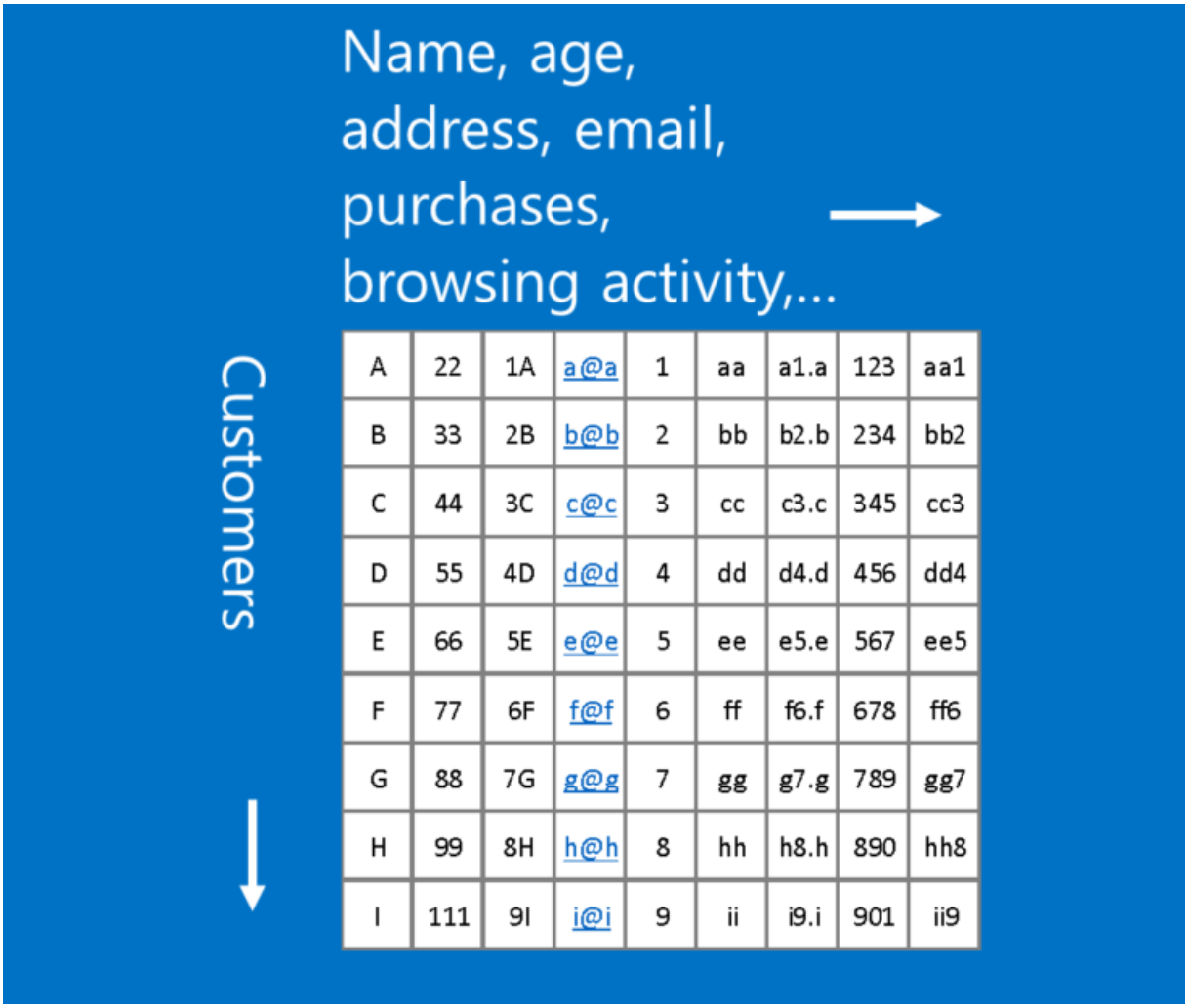
In addition to these there are also higher level architectural decisions to make: How many of each layer to include? In what order? Some deep neural networks can have over a thousand layers, which opens up a lot of possibilities.

With so many combinations and permutations, only a small fraction of the possible CNN configurations have been tested. CNN designs tend to be driven by accumulated community knowledge, with occasional deviations showing surprising jumps in performance. And while we've covered the building blocks of vanilla CNNs, there are lots of other tweaks that have been tried and found effective, such as new layer types and more complex ways to connect layers with each other.

Beyond images



While our X and O example involves images, CNNs can be used to categorize other types of data too. The trick is, whatever data type you start with, to transform it to make it look like an image. For instance, audio signals can be chopped into short time chunks, and then each chunk broken up into bass, midrange, treble, or finer frequency bands. This can be represented as a two-dimensional array where each column is a time chunk and each row is a frequency band. “Pixels” in this fake picture that are close together are closely related. CNNs work well on this. Researchers have gotten quite creative. They have adapted text data for natural language processing and even chemical data for drug discovery.



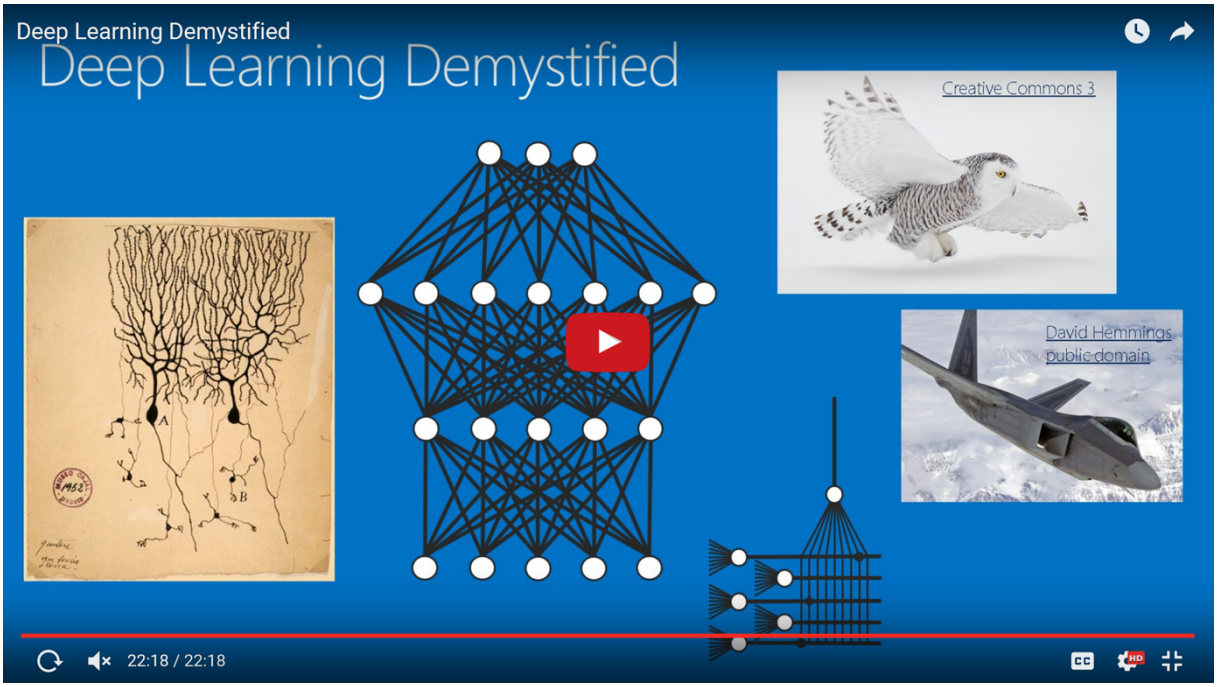
A	22	1A	a@a	1	aa	a1.a	123	aa1
B	33	2B	b@b	2	bb	b2.b	234	bb2
C	44	3C	c@c	3	cc	c3.c	345	cc3
D	55	4D	d@d	4	dd	d4.d	456	dd4
E	66	5E	e@e	5	ee	e5.e	567	ee5
F	77	6F	f@f	6	ff	f6.f	678	ff6
G	88	7G	g@g	7	gg	g7.g	789	gg7
H	99	8H	h@h	8	hh	h8.h	890	hh8
I	111	9I	i@i	9	ii	i9.i	901	ii9

An example of data that doesn't fit this format is customer data, where each row in a table represents a customer, and each column represents information about them, such as name, address, email, purchases and browsing history. In this case, the location of rows and columns doesn't really matter. Rows can be rearranged and columns can be re-ordered without losing any of the usefulness of the data. In contrast, rearranging the rows and columns of an image makes it largely useless.

A rule of thumb: If your data is just as useful after swapping any of your columns with each other, then you can't use Convolutional Neural Networks.

However if you can make your problem look like finding patterns in an image, then CNNs may be exactly what you need.

Learn more



For more details on how to build this X-and-O example, check out Dr. Alexander Hanuschkin's excellent MATLAB and Caffe implementations for NVIDIA GPUs. If you'd like to dig deeper into deep learning, check out my Demystifying Deep Learning post. I also recommend the notes from the Stanford CS 231 course by Justin Johnson and Andrej Karpathy that provided inspiration for this post, as well as the writings of Christopher Olah, an exceptionally clear writer on the subject of neural networks.

If you are one who loves to learn by doing, there are a number of popular deep learning tools available. Try them all! And then tell us what you think.

- Caffe
- CNTK
- Deeplearning4j
- TensorFlow
- Theano
- Torch
- Many others

I hope you've enjoyed our walk through the neighborhood of Convolutional Neural Networks. Feel free to start up a conversation.

Brandon