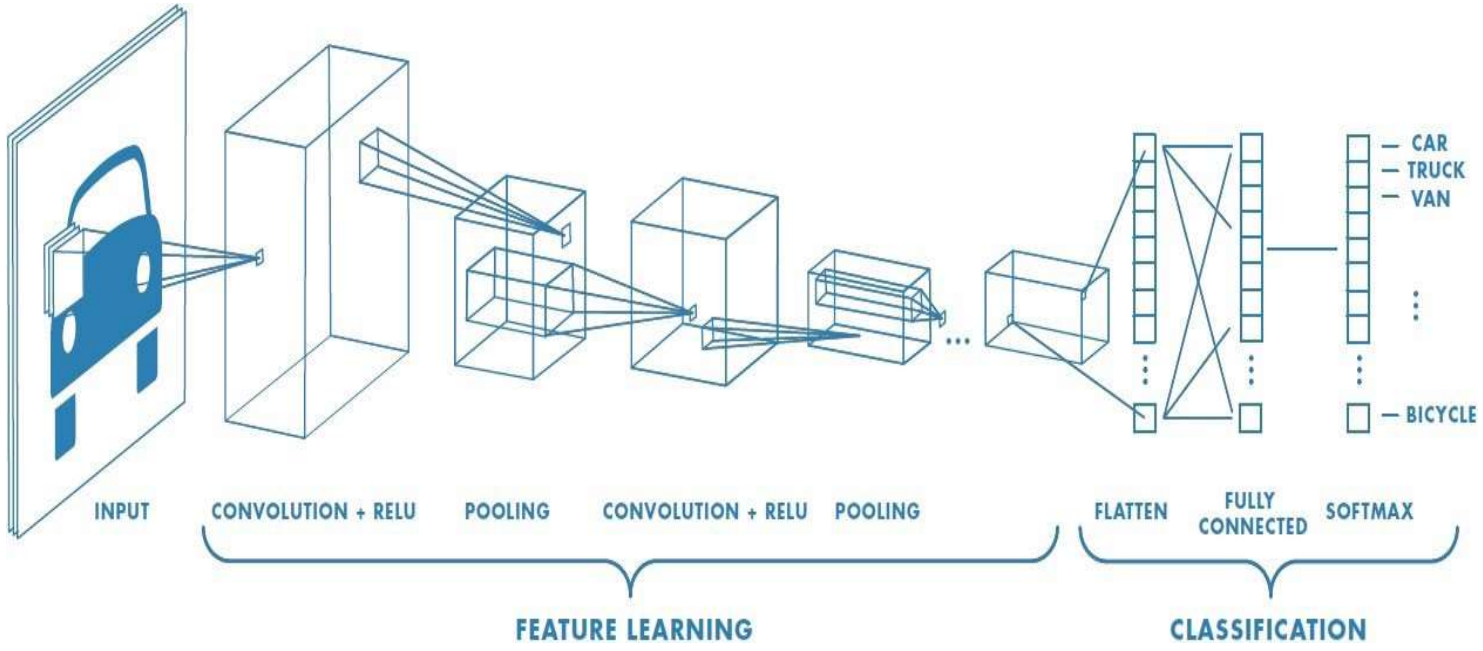
7. Convolutional Neural Networks and Transfer Learning for Image Classification



7.1 Introduction to Convolutional Neural Networks (CNNs)

Convolutional Neural Networks are very similar to other neural networks we have studied in the previous parts. They all are made up of neurons with learnable weights and biases. Each neuron receives inputs, performs a linear operation of the inputs and then a non-linearity operation as the output.

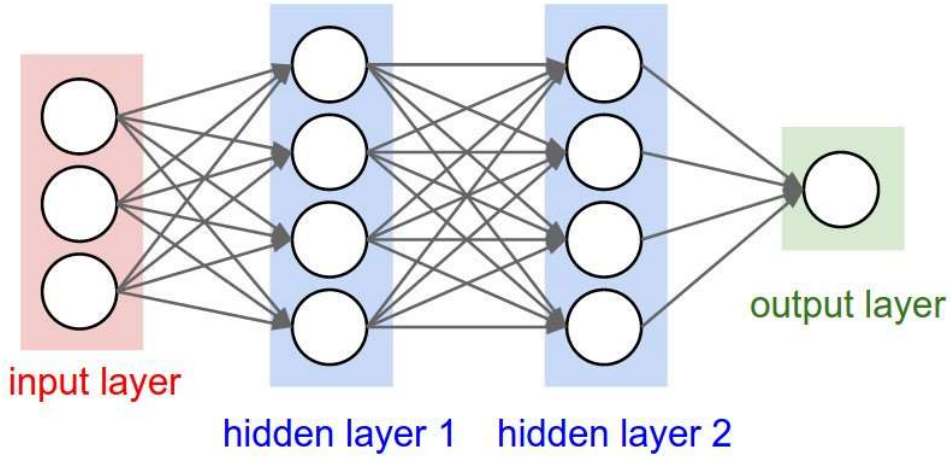
So what are the changes in CNNs?

CNN make the explicit assumption that the inputs are images (or text, but we focus on images in this course), which allows us to encode certain properties into the architecture. These allow it more efficient to implement and vastly reduce the amount of parameters in the network.

Architecture Overview

Regular neural networks

As introduced in previous parts, a neural network receives an input (a single vector), and transform it through a series of *hidden layers*. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer. The last layer is the output layer whose outputs are the class scores in the context of pattern classification.



(i) Regular neural networks do not scale well to full images

In the part of MLP neural network, we introduced one application example, that is the recognition of handwritten digits, where each digit is a grey scale image of size 28x28 pixels. Therefore, each digit image is converted into a 784-dimensional vector and we need to use 784 neurons in the input layer of the MLP neural network.

For colour images, for example, of size 32x32x3 (32 wide, 32 high, 3 color channels), then the vector dimension will be 32x32x3=3072. Thus, a single fully-connected neuron in the first hidden layer of the MLP neural network will have 3072 weights.

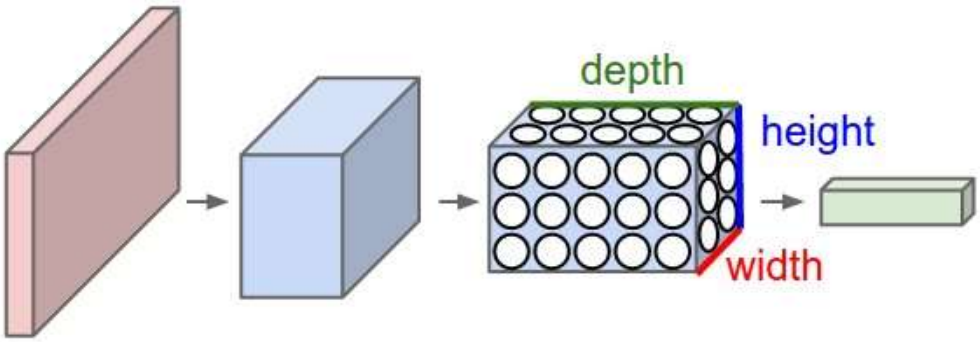
This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g. 200x200x3, would make each neuron in the first hidden layer have 200x200x3 = 120,000 weights.

(2) Regular neural networks can easily lead to overfitting

We certainly will have more than one neurons in the hidden layer, thus, the number of weights (parameters) would add up quickly, and the huge number of parameters would quickly lead to ***overfitting***.

Convolutional neural networks (CNN)

Convolutional neural networks take the advantage of the fact that the input consists of images, and constrain the architecture in a more sensible way. The layers of a CNN have neurons arranged in 3 dimensions: *width, height, depth*.



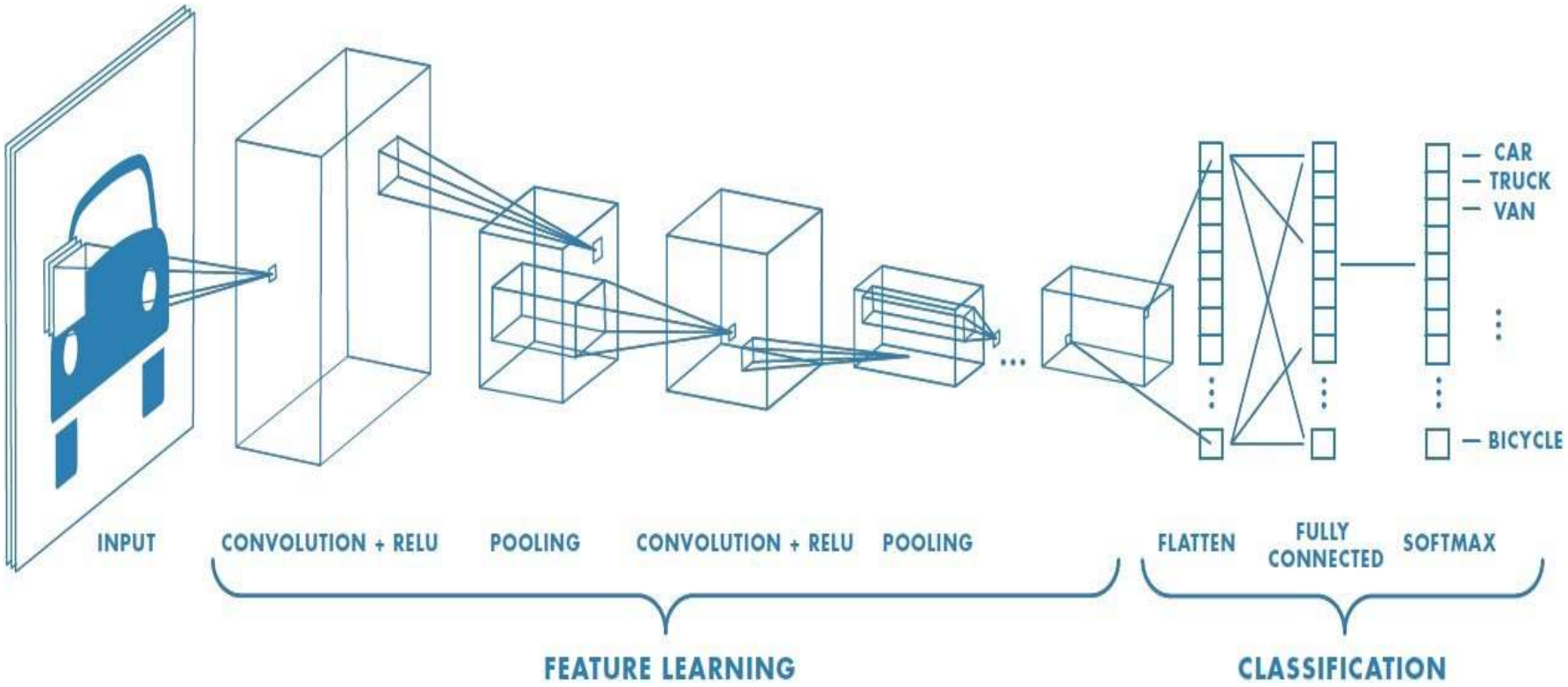
Note that the word *depth* here refers to the third dimension of the image volume, not the depth of a full CNN. For example, if the input image is a colour image with RGB (red, green and blue) three channels, then the volume has dimensions 32x32x3 (width, height, depth respectively). If the image is a grey-scale image, the depth is 1.

As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner.

7.2 Layers Used to Build CNN

As can be seen, a simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of input to another. Three main types of layers are used to build CNN architecture, including

(i) Convolutional Layer
(ii) Pooling Layer
(iii) Fully-Connected Layer (exactly as seen in regular neural networks).



The architecture of a typical convolutional neural network

We next introduce each of the three types of layers.

(i) Convolutional Layer

The Convolution layer is the core building block of a CNN. This layer consists of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a CNN might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional output (activation map) that gives the responses of that filter at every spatial position.

Each convolutional layer has a set of filters, and each of them will produce a separate 2-dimensional activation map. All these activation maps along the depth dimension are stacked to produce the output volume.

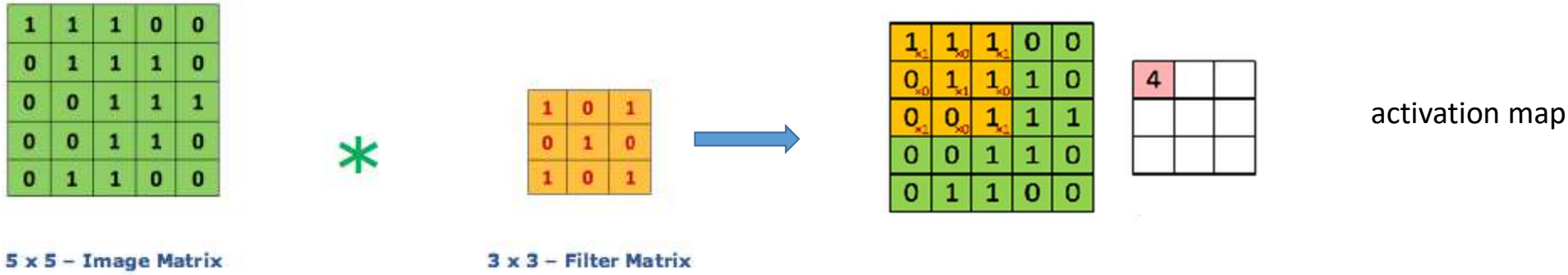
The convolution is much like the following scenario.

In the dark night, we want to find what are on a sheet of bubble wrap using a flashlight. Assuming that your flashlight shines a 5-bubble x 5-bubble area. To look at the entire sheet, you would slide your flashlight across each 5x5 square until you have seen all the bubbles.

The light from the flashlight here is your *filter* and the region you are sliding over is the *receptive field*. The light sliding across the receptive fields is your flashlight *convolving*. Your filter is an array of numbers (also called weights or parameters).

The depth of the filter has to be the same as the depth of the input, so if we were looking at a color image, the depth would be 3. That makes the dimensions of this filter 5x5x3.

In each position, the filter multiplies the values in the filter with the original values in the pixel. This is element wise multiplication. The multiplications are summed up, creating a single number. The array you end up with is called a *feature map* or an *activation map*.

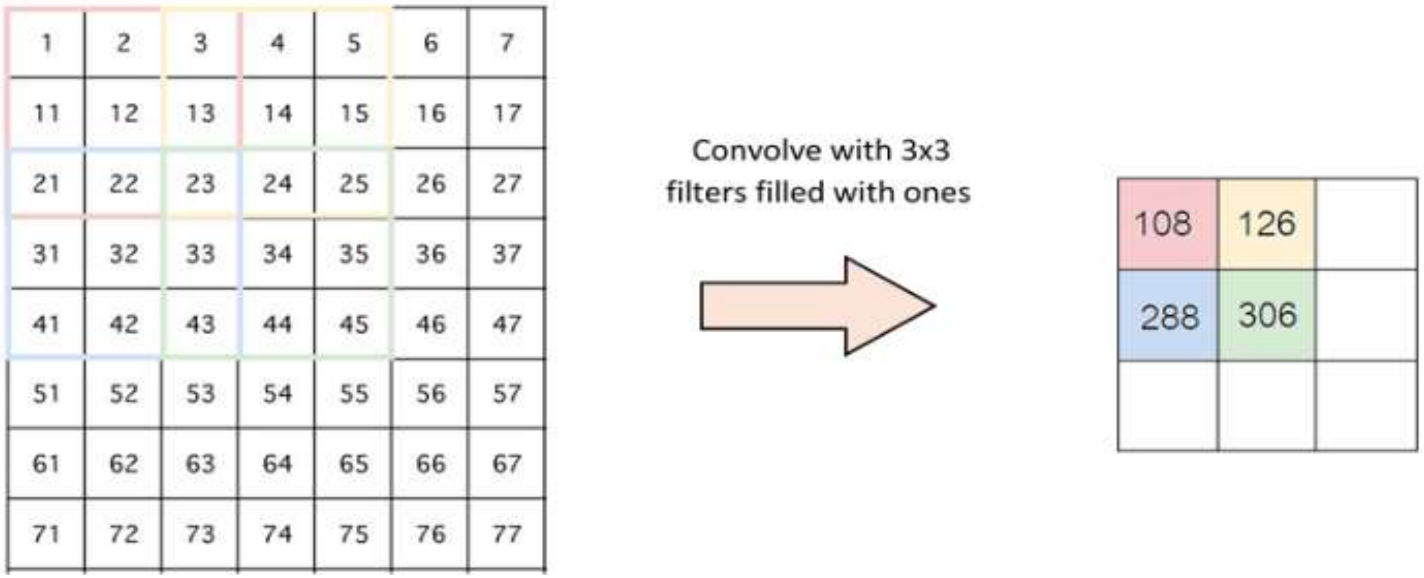


Please open to the following link to see the animation of convolution:

https://miro.medium.com/max/268/1*MrGSULUtkXc0Ou07QouV8A.gif

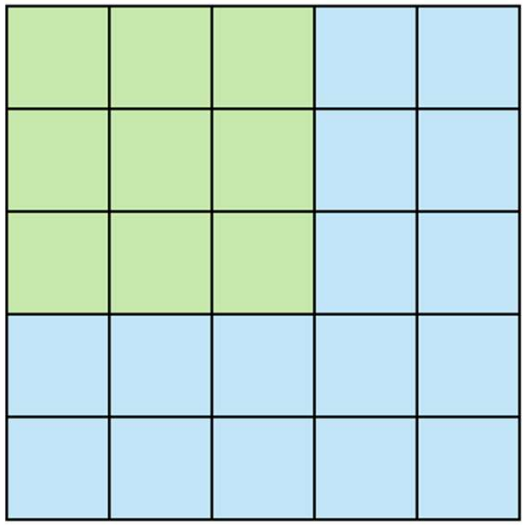
Stride

Stride is the number of pixels shifts over the input matrix. When the stride is 1, then we move the filters 1 pixel at a time. If the stride is set to 2, then we move the filters 2 pixels at a time and so on. The below figure shows convolution working with a stride of 2.

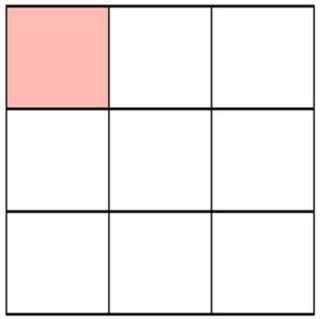


Padding

We can observe that the size of output is smaller than the input after convolution operation even if a stride of 1 is used as shown below.

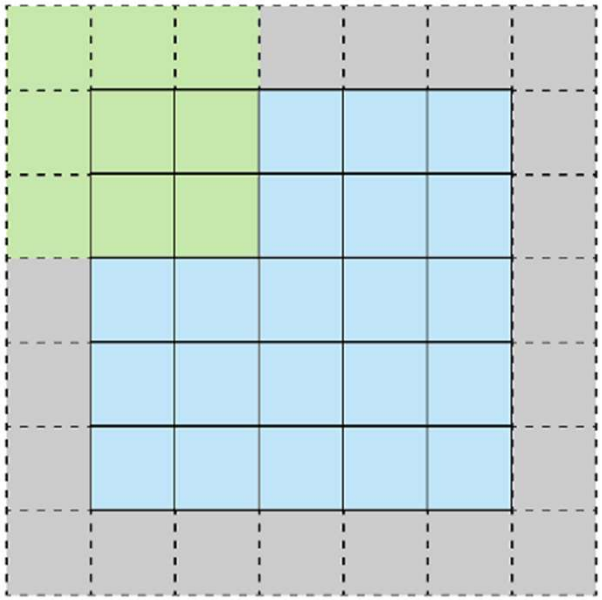


Stride 1

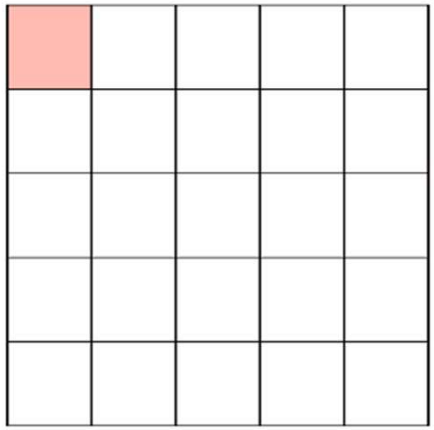


Feature Map

To maintain the dimension of output the same as that of the input , we use padding. Padding is a process of adding zeros to the input matrix symmetrically. As shown in the following example, the extra grey blocks denote the padding.



Stride 1 with Padding



Feature Map

(ii) Pooling Layer

Pooling layers would reduce the number of parameters when the images are too large. Spatial pooling, also called subsampling or downsampling, aims to reduce the dimensionality of each map but retain the important information. Spatial pooling can be of different ways:

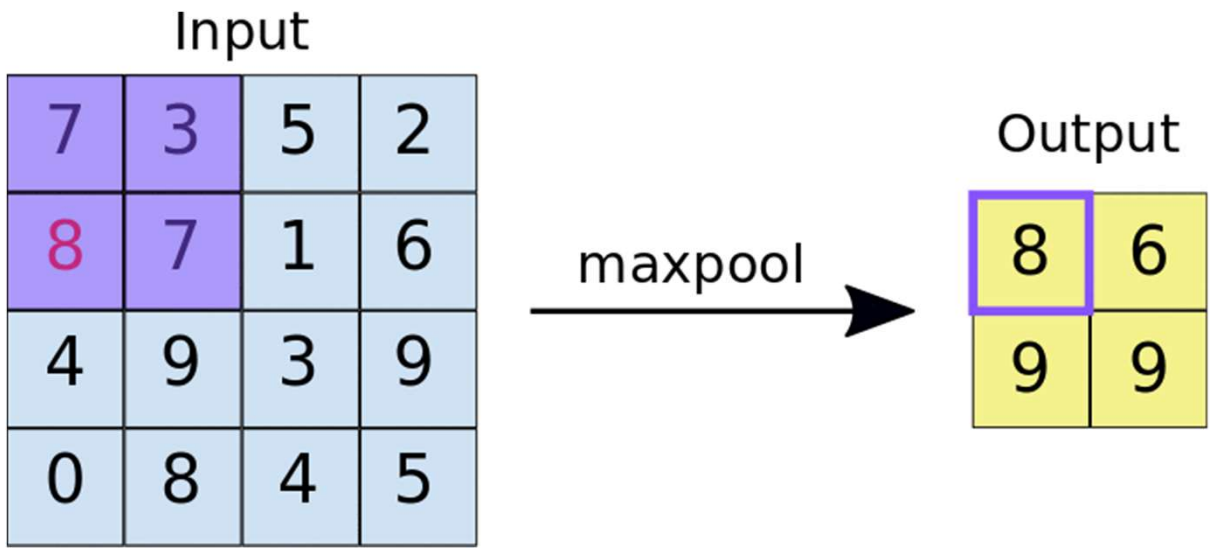
(i) Max pooling
(ii) Average pooling
(iii) Sum pooling

Max pooling

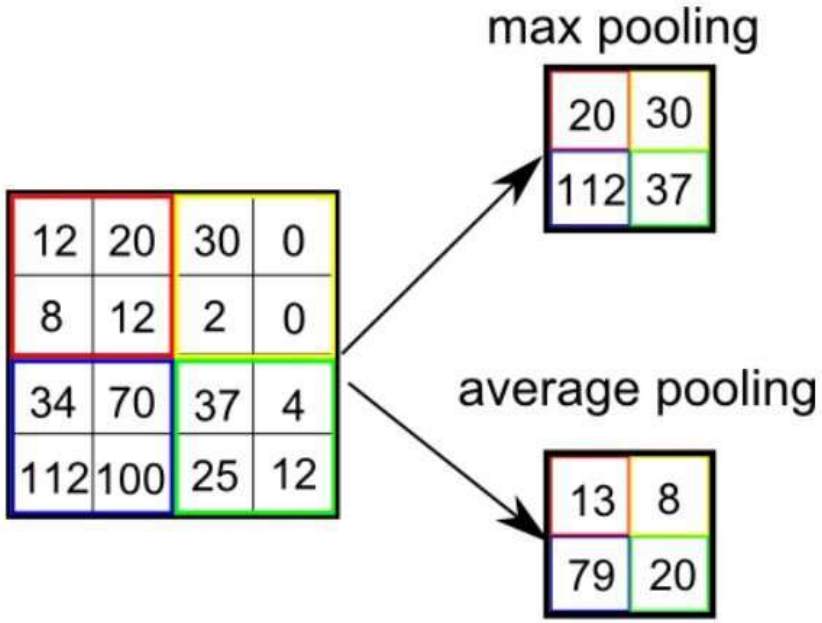
Max pooling takes the largest element in the masked regions of the feature map.

Average and sum pooling

Max pooling and sum pooling takes the average or sum of all elements in the masked regions of the feature map.



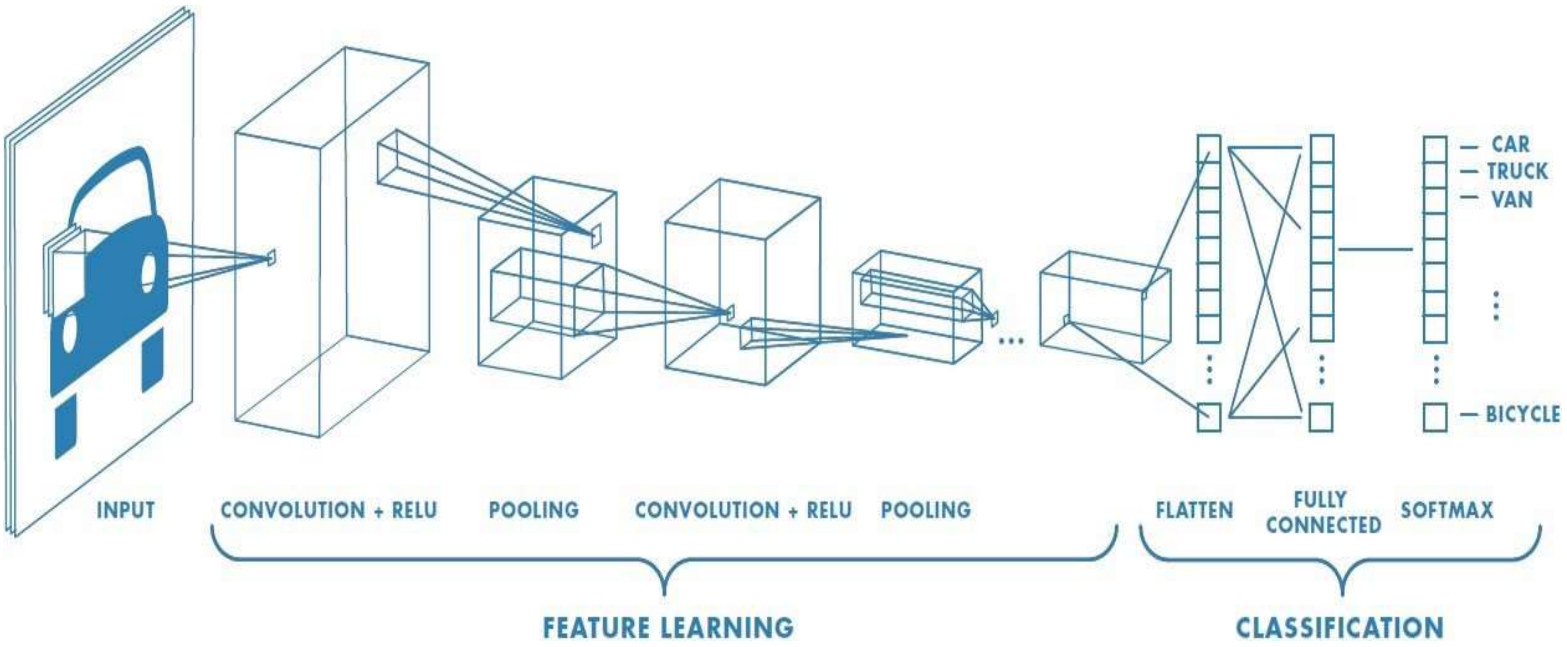
Obviously, the pooling operation reduces the dimensionality significantly: from 4*4 to 2*2.



Max Pooling also has the function of noise suppressing. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. On the other hand, average pooling simply performs dimensionality reduction. Hence, max pooling outperforms average pooling.

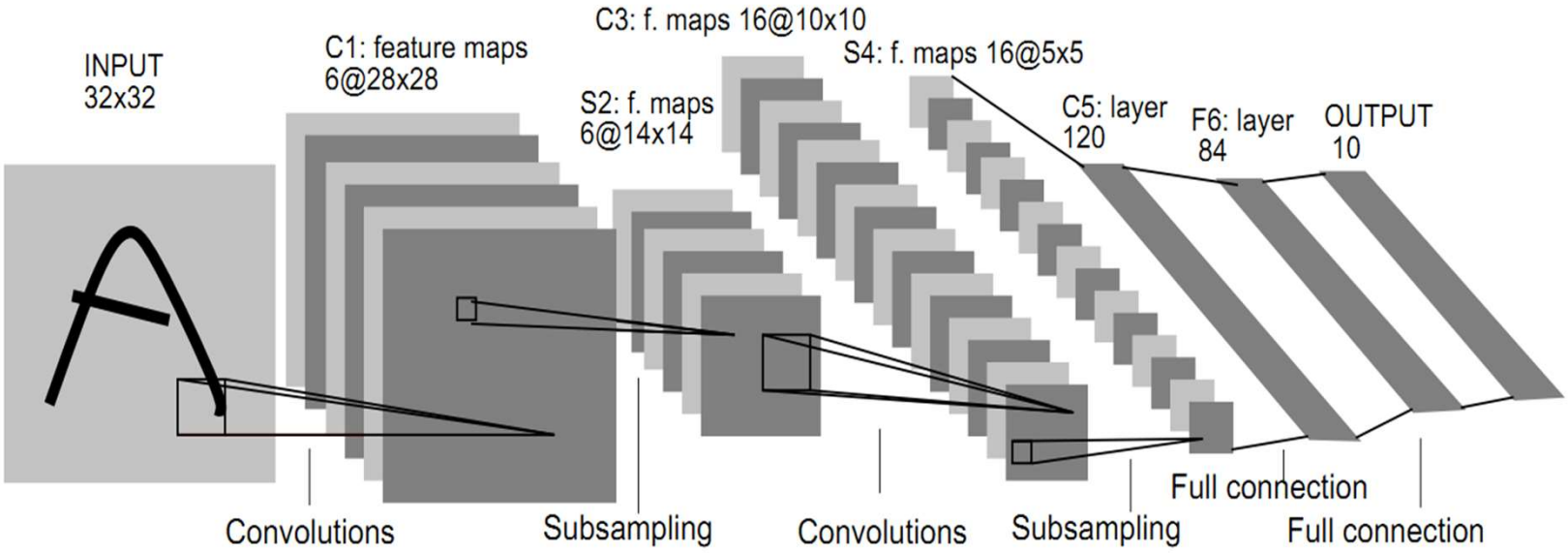
(iii) Fully connected layer

Finally, after several convolutional and max pooling layers, the feature maps are flattened to produce a vector representation of the image features in the fully connected layer. In addition, the fully connected layers also perform pattern classification as shown below:



7.3 Some Popular CNN Architectures

(i) LeNet---LeCun et al



Let's understand the architecture of the model. The model contained 7 layers excluding the input layer. Since it is a relatively small architecture, let's go layer by layer:

Layer 1:

A convolutional layer with filter (kernel) size of 5x5, stride of 1x1 and 6 filters in total. So the input image of size 32x32x1 gives an output of 28x28x6. Total number of weights in this layer is 5x5x6 + 6 (bias terms)

Layer 2:

A pooling layer with 2×2 kernel size, stride of 2x2 and 6 kernels in total. This pooling layer acted a little differently from what we discussed in previous post. The input values in the receptive were summed up and then were multiplied to a trainable parameter (1 per filter), the result was finally added to a trainable bias (1 per filter). Finally, sigmoid activation was applied to the output. So, the input from previous layer of size 28x28x6 gets sub-sampled to 14x14x6. Total weights in layer is [1 (trainable parameter) + 1 (trainable bias)] x 6 = 12

Layer 3:

Similar to Layer 1, this layer is a convolutional layer with the same configuration except that it has 16 filters instead of 6. So the input from previous layer of size 14x14x6 gives an output of 10x10x16. Total weights in this layer is 5x5x16 + 16 = 416.

Layer 4:

Again, similar to Layer 2, this layer is a subsampling layer with 16 filters this time around. Remember, the outputs are passed through sigmoid activation function. The input of size 10x10x16 from previous layer gets sub-sampled to 5x5x16. Total weights in this layer is (1 + 1) x16 = 32.

Layer 5:

This time around we have a convolutional layer with 5×5 kernel size and 120 filters. There is no need to even consider strides as the input size is 5x5x16 so we will get an output of 1x1x120. Total weights in this layer is 5x5x120 = 3000.

Layer 6:

This is a dense layer with 84 parameters. So, the input of 120 units is converted to 84 units. Total weights is 84x120 + 84 = 10164. The activation function used here was rather a unique one.

Output Layer:

Finally, a dense layer with 10 units is used. Total weights is 84 x10 + 10 = 924.

(ii) AlexNet --- Krizhevsky et al

AlexNet is the name of a convolutional neural network that has had a large impact on machine learning, specifically in the application of deep learning to machine vision. It won the 2012 ImageNet LSVRC-2012 competition by a large margin

The architecture consists of 5 Convolutional Layers and 3 Fully Connected Layers. These 8 layers combined with some new concepts at that time

(i) Max Pooling

(ii) ReLU activation function

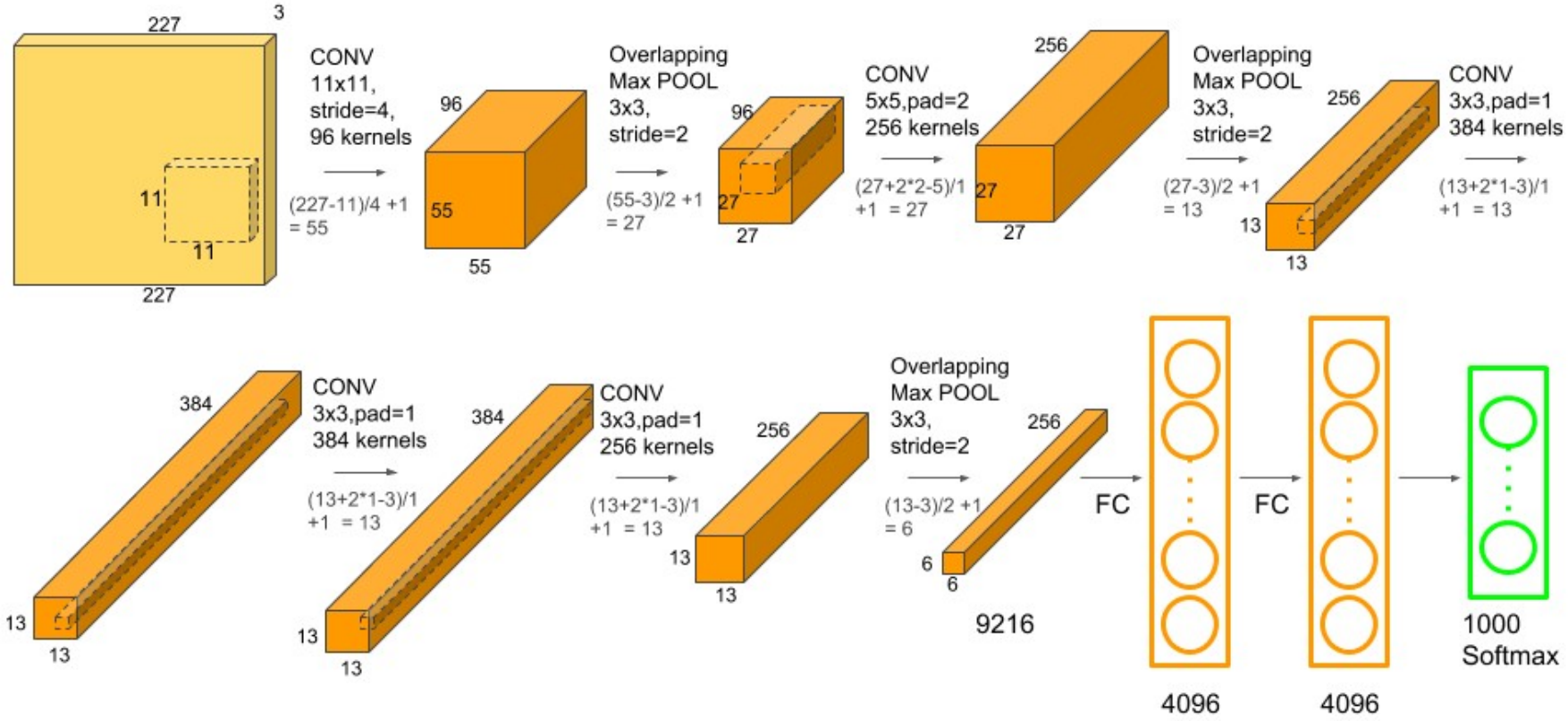
(iii) "Dropout" as the regularization

AlexNet

(i) Has 60 million parameters and 650,000 neurons

(ii) Trained on 1.2 million high-resolution images

(iii) Samples from1000 classes


227
227
3
11
11
CONV
11x11,
stride=4,
96 kernels
(227-11)/4 +1
= 55
55
55
96
Overlapping
Max POOL
3x3,
stride=2
(55-3)/2 +1
= 27
27
27
96
CONV
5x5,pad=2
256 kernels
(27+2*2-5)/1
+1 = 27
27
27
256
Overlapping
Max POOL
3x3,
stride=2
(27-3)/2 +1
= 13
13
13
256
CONV
3x3,pad=1
384 kernels
(13+2*1-3)/1
+1 = 13
13
13
384
CONV
3x3,pad=1
384 kernels
(13+2*1-3)/1
+1 = 13
13
13
384
CONV
3x3,pad=1
256 kernels
(13+2*1-3)/1
+1 = 13
13
13
256
Overlapping
Max POOL
3x3,
stride=2
(13-3)/2 +1
= 6
6
6
256
9216
FC
4096
FC
4096
1000
Softmax

(iii) VGGNet ---Simonyan et al

The runner up of 2014 ImageNet challenge is named VGGNet. Because of the simplicity of its uniform architecture, it appeals to a new-comer as a simpler form of a deep convolutional neural network.

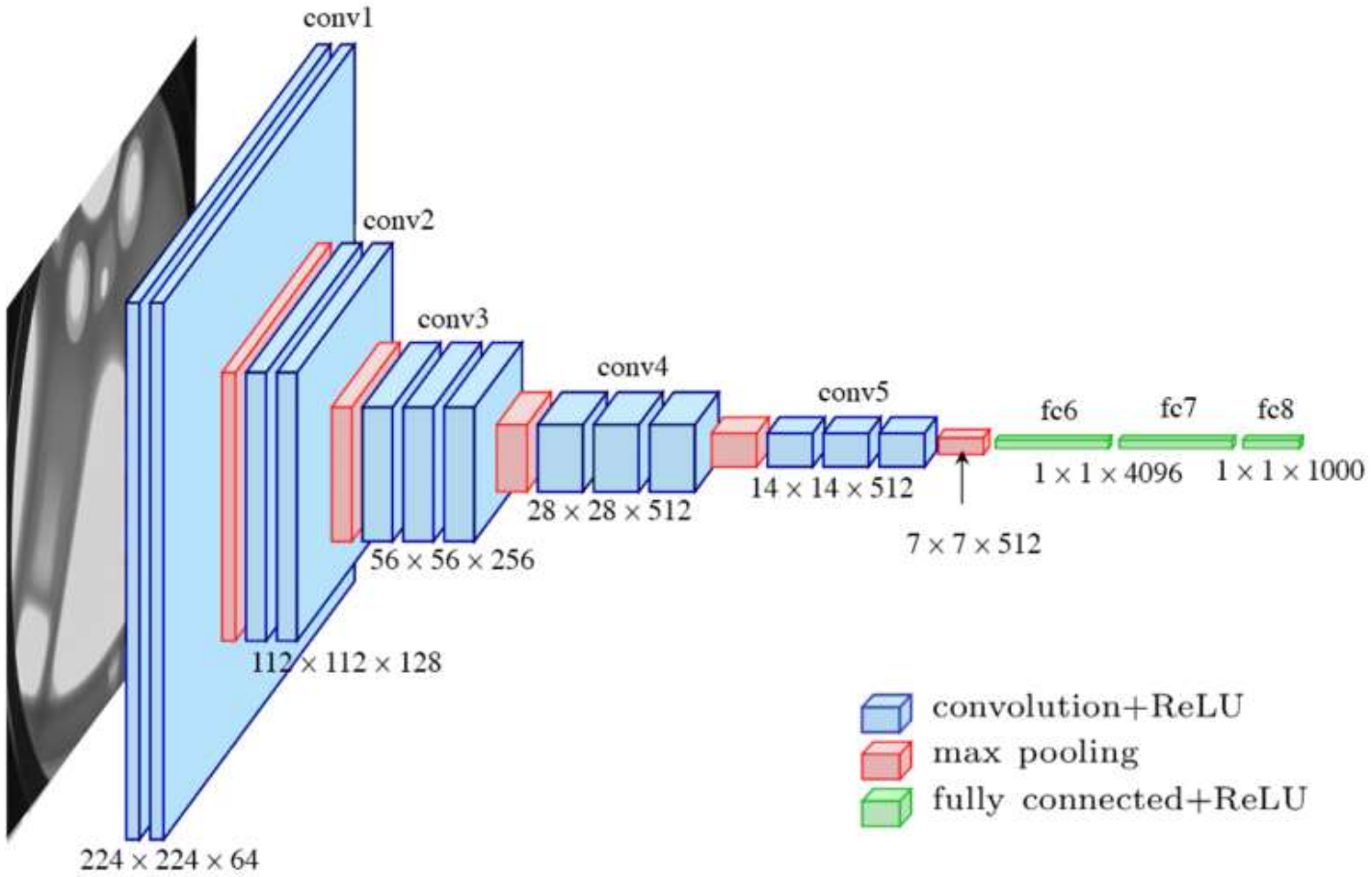
Features

(i) Each Convolutional layer has configuration — kernel size = 3×3, stride = 1×1, The only thing that differs is number of filters.

(ii) Each Max Pooling layer has configuration — windows size = 2×2 and stride = 2×2. Thus, we half the size of the image at every Pooling layer.

(iii) The input is RGB image of 224x224 pixels. So input size = 224x224x3

(iv) Total number of weights is 138 million. Most of these parameters are contributed by fully connected layers.

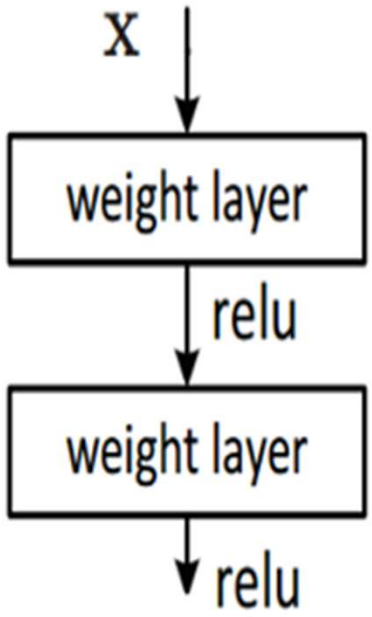

conv1
conv2
conv3
conv4
conv5
fc6
fc7
fc8
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
convolution+ReLU
max pooling
fully connected+ReLU

(iv) ResNet--- Kaiming He et al

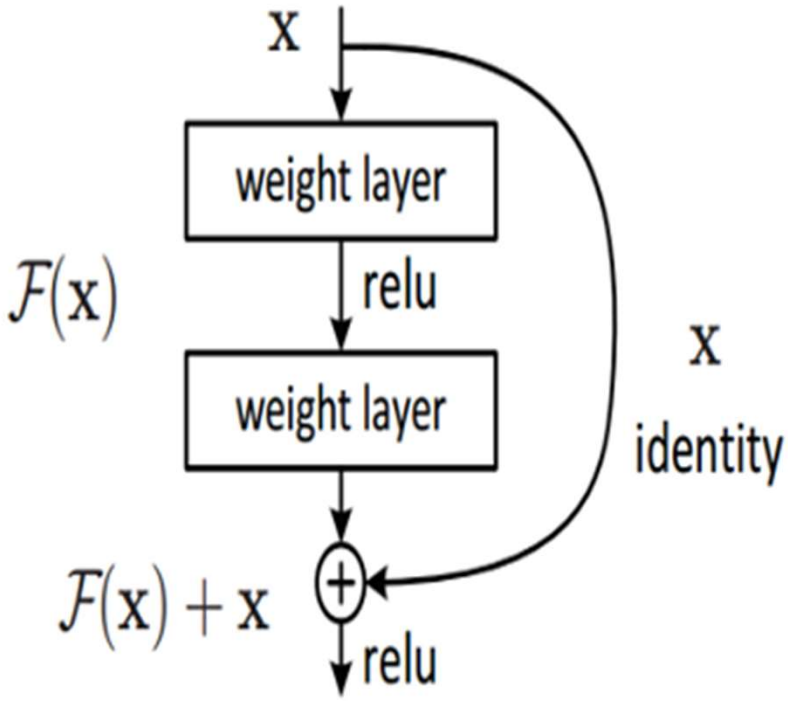
Since AlexNet, the state-of-the-art CNN architecture is going deeper and deeper. While AlexNet has only 5 convolutional layers, the VGG network has 19 layers.

However, increasing network depth does not work by simply stacking layers together. Deep networks are hard to train because of the notorious vanishing gradient problem as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient infinitively small. As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly.

The core idea of ResNet is introducing a so-called “identity shortcut connection” that skips one or more layers, as shown in the following figure. The ResNet architecture which was introduced by Microsoft, won the ILSVRC (ImageNet Large Scale Visual Recognition Challenge) in 2015.



Conventional method



ResNet method

7.4 Training of CNN

Training of CNN is based on gradient descent optimization, similar to MLP neural networks. But the mini-batch mode is adopted. Under this mode, the gradient descent is the average over the mini-batch, and is called stochastic gradient descent (SGD).

Different CNNs adopt different weight updating rules. Take the AlexNet as an example. The batch size of 128 samples, momentum of 0.9, and weight decay of 0.0005 are introduced into weight updating rule:

$$\Delta \mathbf{w}(n) = 0.9 \times \Delta \mathbf{w}(n-1) + 0.0005 \times \eta \times \mathbf{w}(n-1) - \eta \times \left\langle \frac{\partial J}{\partial \mathbf{w}} \Big|_{\mathbf{w}=\mathbf{w}(n-1)} \right\rangle_{D_n}$$

where $\langle \frac{\partial J}{\partial \mathbf{w}} |_{\mathbf{w}=\mathbf{w}(n-1)} \rangle_{D_n}$ denotes the average over the n-th batch D_n of the derivative of the objective with respect to **w**, evaluated at **w**(n-1).

7.5 Transfer Learning in CNN

Humans have an inherent ability to transfer knowledge across tasks. What we acquire as knowledge while learning about one task, we utilize in the same way to solve related tasks. The more related the tasks, the easier it is for us to transfer, or cross-utilize our knowledge. Some simple examples are:

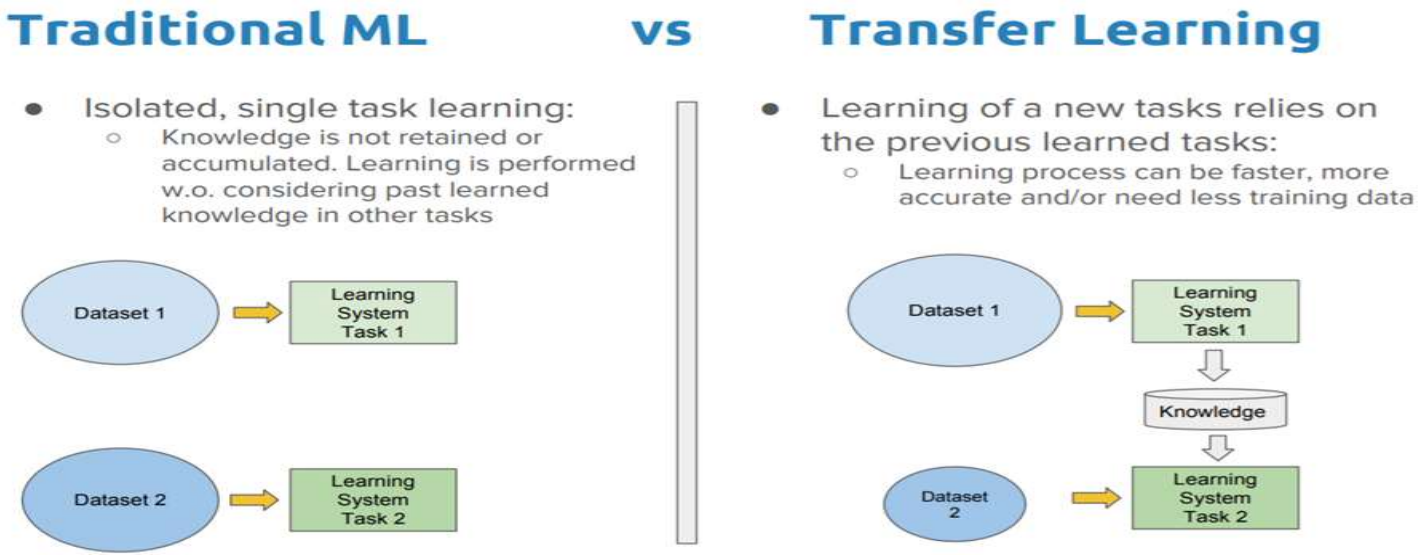
(i) Know how to ride a motorbike ➦ Learn how to ride a car

(ii) Know how to play classic piano ➦ Learn how to play jazz piano

(iii) Know math and statistics ➦ Learn machine learning

In each of the above scenarios, we do not learn everything from scratch when we attempt to learn new aspects or topics. We transfer and leverage our knowledge from what we have learnt in the past!

Conventional machine learning and deep learning algorithms have been traditionally designed to work in isolation. These algorithms are trained to solve specific tasks. The models have to be rebuilt from scratch once the feature-space distribution changes. Transfer learning is the idea of overcoming the isolated learning paradigm and utilizing knowledge acquired for one task to solve related ones.



Three important questions of transfer learning

To use transfer learning, the following three important questions must be answered:

(i) What to transfer?

This is the first and the most important step in the whole process. We try to seek answers about which part of the knowledge can be transferred from the source to the target in order to improve the performance of the target task. We should try to identify which portion of knowledge is source-specific and what is common between the source and the target.

(ii) When to transfer?

There can be scenarios where transferring knowledge for the sake of it may make matters worse than improving anything (also known as negative transfer). We should aim at utilizing transfer learning to improve target task performance/results and not degrade them. We need to be careful about when to transfer and when not to.

(iii) How to transfer?

Once the *what* and *when* have been answered, we can proceed towards identifying ways of actually transferring the knowledge across domains/tasks. This involves changes to existing algorithms and different techniques.

Three major transfer learning scenarios

It is common to pre-train a CNN on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the CNN either as an initialization or a fixed feature extractor for the task of interest.

The three major transfer learning scenarios are as follows:

(i) CNN as fixed feature extractor

Take a CNN pre-trained on ImageNet, remove the last fully-connected layer (this layer's outputs are the 1000 class scores for a different task like ImageNet), then treat the rest of the CNN as a fixed feature extractor for the new dataset.

In an AlexNet, for example, this would compute a 4096-D vector for every image that contains the activations of the hidden layer immediately before the classifier. We call these features *CNN codes*. Once you extract the 4096-D codes for all images, train a linear classifier (e.g. Linear SVM or Softmax classifier) for the new dataset.

(ii) Fine-tuning the CNN

The second strategy is not only to replace and retrain the classifier on top of the CNN on the new dataset, but also to fine-tune the weights of the pre-trained network by continuing the backpropagation.

It is possible to fine-tune all the layers of the CNN, or it is possible to keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network. This is motivated by the observation that the earlier features of a CNN contain more generic features (e.g. edge detectors or color blob detectors) that should be useful to many tasks, but later layers of the CNN becomes progressively more specific to the details of the classes contained in the original dataset.

In case of ImageNet, for example, which contains many dog breeds, a significant portion of the representational power of the CNN may be devoted to features that are specific to differentiating between dog breeds.

(iii) Pre-trained models

Since modern CNN take 2-3 weeks to train across multiple GPUs on ImageNet, it is common to see people release their final CNN checkpoints for the benefit of others who can use the networks for fine-tuning.

When and how to fine-tune?

How do you decide what type of transfer learning you should perform on a new dataset? This is a function of several factors, but the two most important ones are

(i) The size of the new dataset (small or big), and

(ii) Its similarity to the original dataset (e.g. ImageNet-like in terms of the content of images and the classes, or very different, such as microscope images).

Keep in mind that CNN features are more generic in early layers and more original-dataset-specific in later layers, here are some common rules of thumb for navigating the 4 major scenarios:

(i) New dataset is small and similar to original dataset.

Since the data is small, it is not a good idea to fine-tune the CNN due to overfitting concerns. Since the data is similar to the original data, we expect higher-level features in the CNN to be relevant to this dataset as well. Hence, the best idea might be to train a linear classifier on the CNN codes.

(ii) New dataset is large and similar to the original dataset

Since we have more data, we can have more confidence that we will not overfit the training data if we try to fine-tune the full network.

(iii) New dataset is small but very different from the original dataset.

Since the data is small, it is likely best to only train a linear classifier. Since the dataset is very different, it might not be best to train the classifier from the top of the network, which contains more dataset-specific features. Instead, it might work better to train the SVM classifier from activations somewhere earlier in the network.

(iv) New dataset is large and very different from the original dataset.

Since the dataset is very large, we may expect that we can afford to train a CNN from scratch. However, in practice it is very often beneficial to initialize with weights from a pre-trained model. In this case, we would have enough data and confidence to fine-tune the entire network.

8. Recurrent Neural Networks for Data Sequence

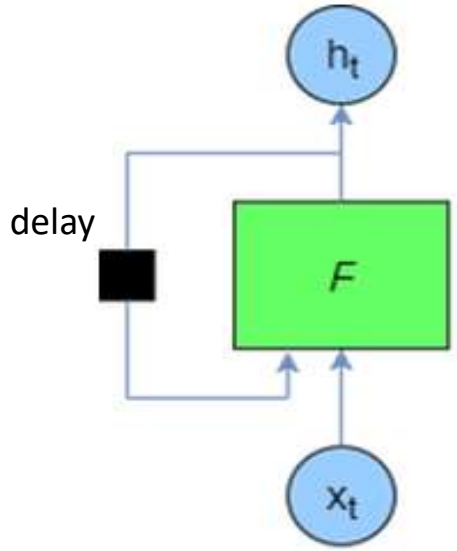
In the part of convolutional neural networks (CNNs), we have learned how to perform classification tasks on static images. However, what happens if we want to analyse dynamic data or sequence data such as video, voice, or text?

There are ways to do some of this using CNNs, but the most popular method of performing classification and other analysis on *sequences* of data is recurrent neural networks (RNNs).

This part will give a brief introduction to recurrent neural networks and a subset of such networks – long-short term memory neural networks (or LSTM).

8.1 An Introduction to Recurrent Neural Networks

A recurrent neural network, at its most fundamental level, is simply a type of densely connected neural network. However, the key difference to normal feed forward networks is the introduction of *time* – in particular, the output of the hidden layer in a recurrent neural network is *fed back to itself* as shown below:

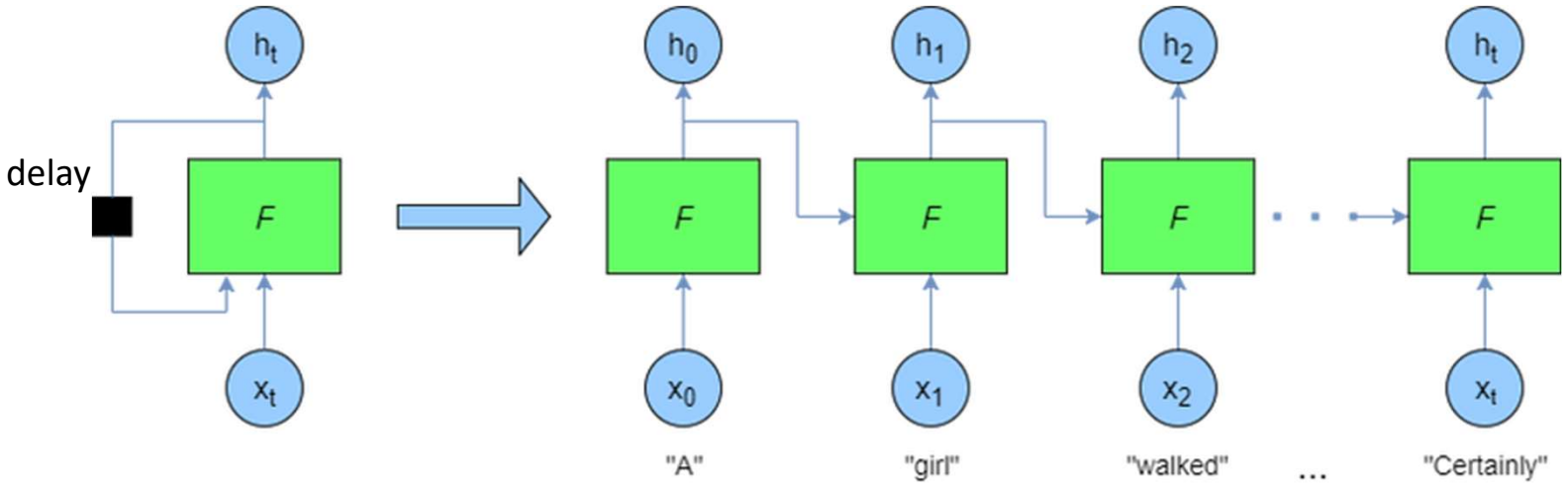


In the diagram above, the input is fed into a hidden layer. The outputs of the hidden layer are passed through a conceptual delay block and fed back into the hidden layer. This means that we can now model *time* or sequence-dependent data.

A good example of this is the prediction of text sequences. Consider the following text string:

> *"A girl walked into a bar, and she said 'Can I have a drink please?'.*
>
> *The bartender said 'Certainly {}".*

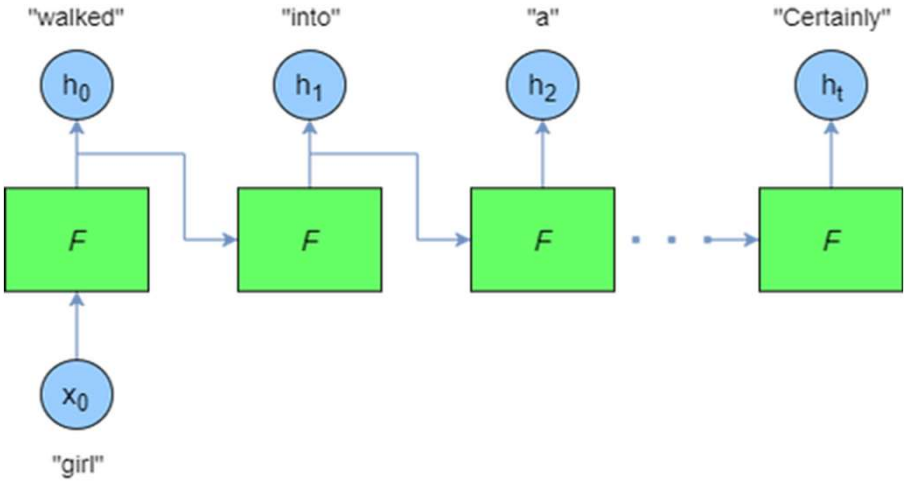
There are many options for what could fill in the {} symbol in the above string, for instance, "miss", "ma'am" and so on. However, other words could also fit, such as "sir", "Mister" etc. In order to get the correct gender of the noun, the neural network needs to "recall" that two previous words designating the likely gender (i.e. "girl" and "she") were used. This type of flow of information through time (or sequence) in a recurrent neural network is shown in the diagram below, which *unrolls* the sequence:



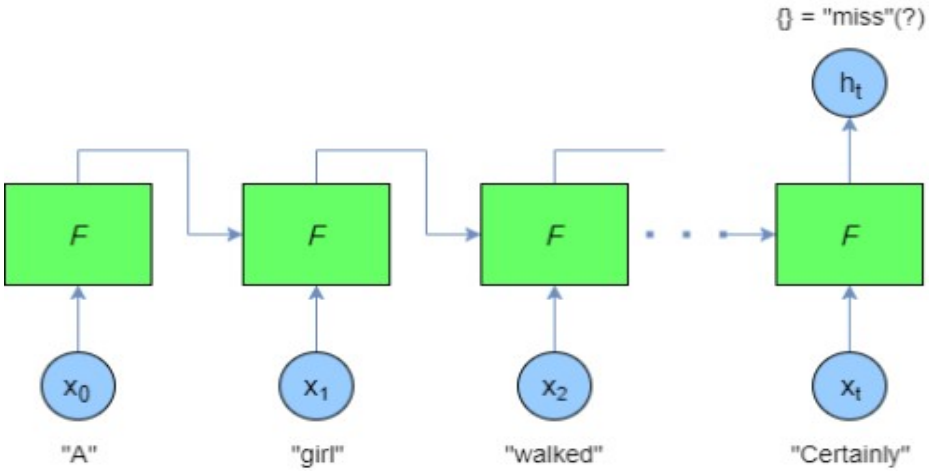
Since only finite length sequences are applied to such networks, we can *unroll* the network as shown on the right-hand side of the diagram above. This unrolled network shows how we can supply a stream of data to the recurrent neural network. For instance, first, we supply the word vector for "A" to the network *F*, the output of the nodes in *F* are fed into the "next" network and also act as a stand-alone output h_0. The next network *F* (actually the same network) at time *t=1* takes the next word vector for "girl" and the previous output h_0 into its hidden nodes, producing the next output h_1 and so on.

Please note, the words themselves i.e. “A”, “girl” etc. are not input directly into the neural network. An embedding vector is used for each word. An embedding vector is an efficient vector representation of the word (often between 50-300 in length), which should maintain some meaning or context of the word. Word embedding is not introduced here, you can search “word2vec” on the web to get a good understanding of it.

Now, back to recurrent neural networks. Recurrent neural networks are very flexible. In the implementation shown above, we have a many-to-many model – in other words, we have the input sequence “A girl walked into a bar…” and many outputs – h_0 to h_t. We could also have multiple other configurations. Another option is one-to-many i.e. supplying one input, say “girl” and predicting multiple outputs h_0 to h_t (i.e. trying to generate sentences based on a single starting word). A further configuration is many-to-one i.e. supplying many words as input, like the sentence “A girl walked into a bar, and she said ‘Can I have a drink please?’. The bartender said ‘Certainly {}” and predicting the next word i.e. {}. The diagram below shows an example one-to-many and many-to-one configuration, respectively (the words next to the outputs are the target words).



one-to-many



many-to-one

Gradient vanishing problem of RNN

Vanilla (naive) recurrent neural networks are not actually used very often in practice. The main reason is the vanishing gradient problem. For recurrent neural networks, ideally, we would want to have long memories, so the network can connect data relationships at significant distances in time. That sort of network could make real progress in understanding how language and narrative works, how stock market events are correlated and so on. However, the more time steps we have, the more chance we have of back-propagation gradients either accumulating and exploding or vanishing down to nothing.

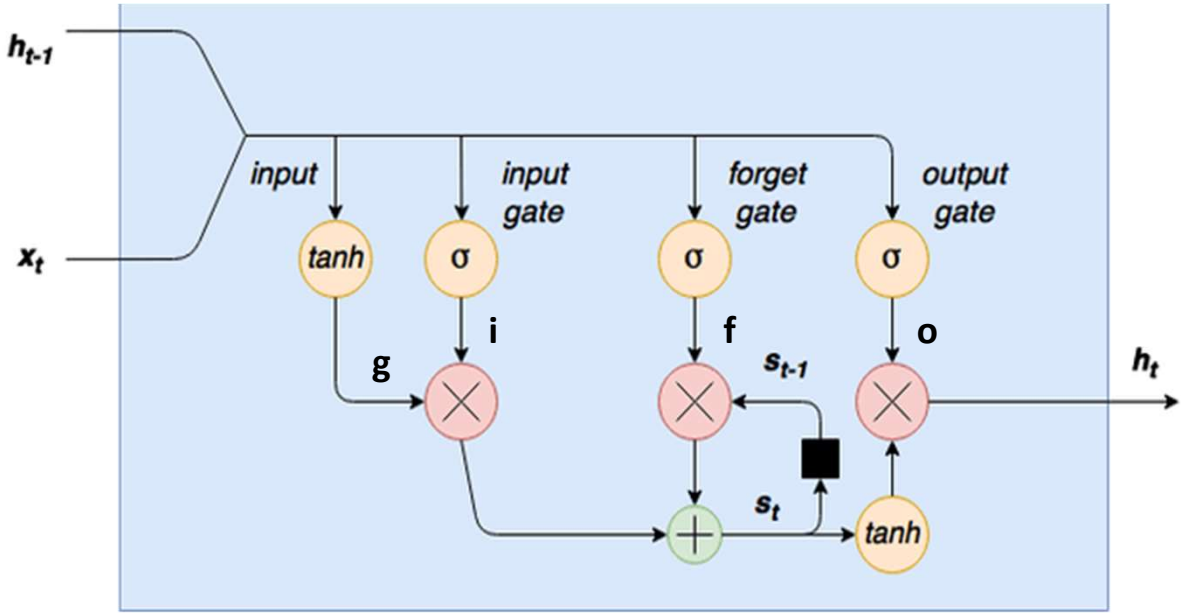
8.2 Long Short-Term Memory (LSTM) Neural Networks

To reduce the vanishing (and exploding) gradient problem, and therefore allow deeper networks and recurrent neural networks to perform well in practical settings, there needs a way to reduce the multiplication of gradients which are less than zero. The LSTM cell is a specifically designed unit of logic that will help alleviate the vanishing gradient problem to make recurrent neural networks more useful for long-term memory tasks i.e. text sequence predictions. The way it does so is by creating an internal memory state which is simply *added* to the processed input, which greatly reduces the multiplicative effect of small gradients.

The time dependence and effects of previous inputs are controlled by an interesting concept called a *forget gate*, which determines which states are remembered or forgotten. Two other gates, the *input gate* and *output gate*, are also featured in LSTM cells.

LSTM Cell

The following diagram show the LSTM cell (i.e. the function *F* in RNN)



The data flow is from left-to-right in the diagram above, with the current input $\mathbf{x}_t$ and the previous cell output $\boldsymbol{h}_{t-1}$ concatenated together to form the top "data rail".

(i) The input gate

First, the input is squashed between -1 and 1 by using a *tanh* activation function. This can be expressed by:

$$\mathbf{g} = \tanh(\mathbf{x}_t \mathbf{W}_x^g + \mathbf{h}_{t-1} \mathbf{W}_h^g + \mathbf{b}^g)$$

Where $\mathbf{W}_x^g$ and $\mathbf{W}_h^g$ are the weights for the input and previous cell output, respectively, and $\mathbf{b}^g$ is the input bias.

The $\mathbf{g}$ is then multiplied element-wise by the output of the *input gate*. The input gate is basically a hidden layer of sigmoid activated nodes, with weighted $\mathbf{x}_t$ and $\boldsymbol{h}_{t-1}$ as input values, and the outputs values of between 0 and 1. The expression for the input gate is:

$$\mathbf{i} = \tanh(\mathbf{x}_t \mathbf{W}_x^i + \mathbf{h}_{t-1} \mathbf{W}_h^i + \mathbf{b}^i)$$

Where $\mathbf{W}_x^i$ and $\mathbf{W}_h^i$ are the weights for the input and previous cell output, respectively, and $\mathbf{b}^i$ is the bias.

The output of the input stage of the LSTM cell is the element-wise product (i.e. multiplication) of **g** and **i** as expressed below:

$$\mathbf{i} * \mathbf{g}$$

Where * denotes element-wise product. The above can be interpreted as that the input gate output **i** acts as the weights for **g**.

(ii) The internal state and forget gate

As can be observed, there is a new variable $\boldsymbol{s}_t$, which is the inner state of the LSTM cell. This state is delayed by one-time step and is ultimately added to the **g*i** input to provide an internal recurrence loop to learn the relationship between inputs separated by time. Two points to note. The first point is the forget gate, which is again a sigmoid activated set of nodes which is element-wise multiplied by $\boldsymbol{s}_{t-1}$ to determine which previous states should be remembered (i.e. forget gate output close to 1) and which should be forgotten (i.e. forget gate output close to 0). This allows the LSTM cell to learn appropriate context.

Consider the sentence "Clare took Helen to Paris and she was very grateful" – for the LSTM cell to learn who "she" refers to, it needs to forget the subject "Clare" and replace it with the subject "Helen". The forget gate can facilitate such operations and is expressed as:

$$\mathbf{f} = \sigma(\mathbf{x}_t \mathbf{W}_x^f + \mathbf{h}_{t-1} \mathbf{W}_h^f + \mathbf{b}^f)$$

The output of the element-wise product of the previous state and the forget gate is expressed as $\mathbf{s}_{t-1} * \mathbf{f}$. Again, the forget gate output acts as weights for internal state.

The second point to note about this stage is that the forget-gate-"filtered" state is simply added to the input, rather than multiplied by it, or mixed with it via weights and a sigmoid activation function as occurs in a standard recurrent neural network. This is important to alleviate the issue of vanishing gradient. The output from this stage is expressed by:

$$\mathbf{s}_t = \mathbf{s}_{t-1} * \mathbf{f} + \mathbf{g} * \mathbf{i}$$

(iii) The output gate

The final stage of the LSTM cell is the output gate. The output gate has two components: another *tanh* squashing function and an output sigmoid gating function. The output sigmoid gating function, like the other gating functions in the cell, is multiplied by the squashed state $\mathbf{s}_t$ to determine which values of the state are output from the cell. As you can tell, the LSTM cell is very flexible, with gating functions controlling what is input, what is "remembered" in the internal state variable, and finally what is output from the LSTM cell.

The output gate is expressed as:

$$\mathbf{o} = \sigma(\mathbf{x}_t \mathbf{W}_x^o + \mathbf{h}_{t-1} \mathbf{W}_h^o + \mathbf{b}^o)$$

The final output of the cell can be expressed as:

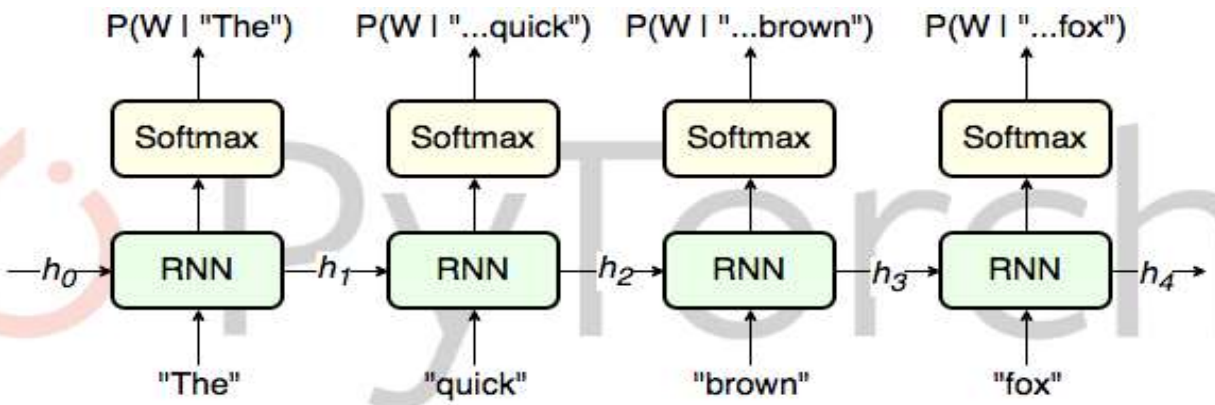
$$\mathbf{h}_t = \tanh(\mathbf{s}_t) * \mathbf{o}$$

8.3 Application Examples of LSTM

LSTM has been used successfully in sequence data, in particular in tasks of natural language processing (i.e. text) . Some examples are:

(i) Language modeling

Language Modelling is the core problem for a number of natural language processing tasks. A trained language model learns the likelihood of occurrence of a word based on the previous sequence of words used in the text



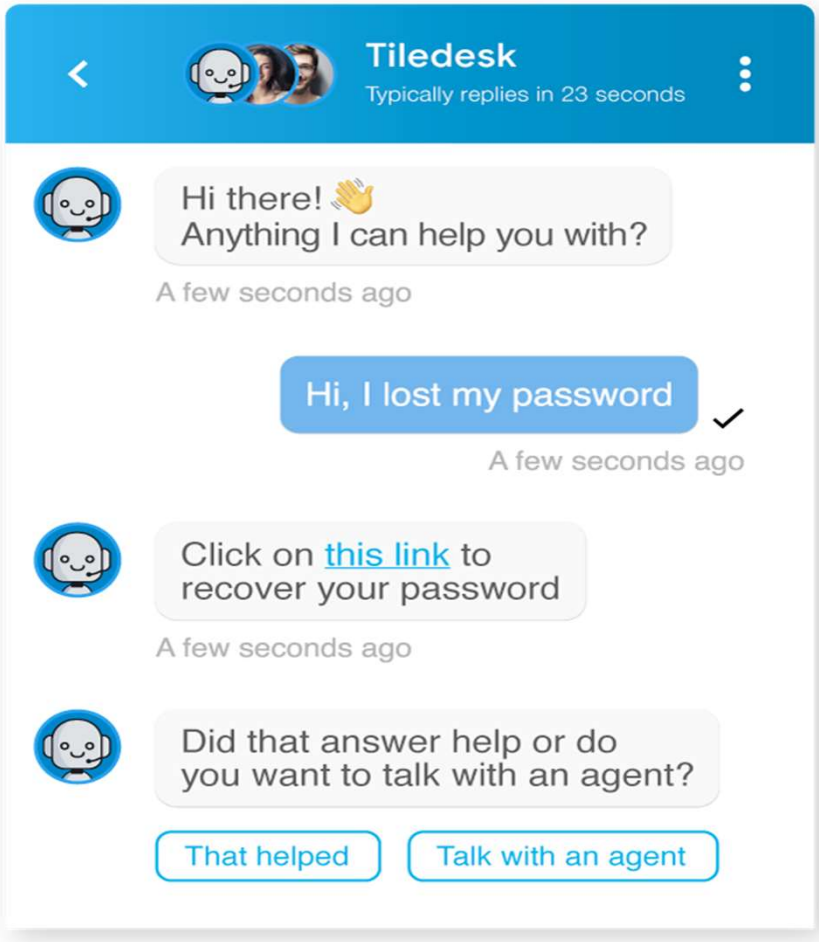
(ii) Machine translation also known as sequence to sequence learning

Machine translation is the translation of text by a computer, with no human involvement. It is also referred to as automated translation or instant translation.

(iii) Image captioning

Image captioning is the process of generating textual description of an image.

(iv) Question answering and chatbot



Thank You!