DEEP LEARNING NEURAL NETWORKS[1]

- Deep neural networks refer to networks multiple hidden layers ($\gg 1$) that allow us to extract complex features and learn complex patterns and associations.
  - For example, for image processing, network can start to learn part-whole decompositions: First layer might learn to group together pixels in an image in order to detect edges. Second layer might then group together edges to detect longer contours, or perhaps detect simple "parts of objects." An even deeper layer might then group together these contours or detect even more complex features.
- It is important to use a non-linear activation function in each hidden layer.
- Formally, one can show that there are functions which a k-layer network can represent compactly (with a number of hidden units that is polynomial in the number of inputs), that a $(k-1)$-layer network cannot represent unless it has an exponentially large number of hidden units.
- By using a deep network, in the case of images, one can also start to learn.
- Finally, cortical computations (in the brain) also have multiple layers of processing.
  - For example, visual images are processed in multiple stages by the brain, by cortical area "V1", followed by cortical area "V2" (a different part of the brain), and so on.

DIFFICULTY TRAINING DEEP ARCHITECTURES

- While the theoretical benefits of deep networks (compactness and expressive power) have been known for many decades, until recently we had little success training deep architectures.
- Applying standard gradient descent methods (e.g., backpropagation algorithm) to try to drive down the training error does not usually work. There are several reasons for this:
  - *Availability of Data*: One normally relies only on labeled data for training. However, labeled data is often scarce, and thus for many problems it is difficult to get enough examples to fit the parameters of a complex model. In fact, given the high degree of expressive power of deep networks, training on insufficient data would also result in overfitting.
  - *Local Optima*: Training a shallow network (with 1 hidden layer) using supervised learning usually resulted in the parameters converging to reasonable values; but when we are training a deep network, this works much less well due bad local optima.
  - *Diffusion of Gradients*: When using backpropagation type algorithms to compute the derivatives, the gradients that are propagated backwards rapidly diminish in magnitude as the depth of the network increases. This is often called the "diffusion of gradients" or “vanishing gradients” problem.

KEY IDEA: GREEDY LAYER-WISE TRAINING

- A key idea to overcome difficulties with deep learning is to *train the layers of the network one at a time*, so that we first train a network with 1 hidden layer, and only after that is done, train a network with 2 hidden layers, and so on.

[1] Primary Source: http://ufldl.stanford.edu

- At each step, we take the old network with $k - 1$ hidden layers, and add an additional k-th hidden layer (that takes as input the previous hidden layer $k - 1$ that we had just trained).
- *Training can either be supervised* (say, with classification error as the objective function on each step), *but more frequently it is unsupervised* (as in an *autoencoder* or *restricted Boltzmann machine*).
- The weights from training the layers individually are then used to initialize the weights in the final/overall deep network, and only then is the entire architecture "fine-tuned" (i.e., trained together to optimize the labeled training set error).
- The success of greedy layer-wise training has been attributed to a number of factors:
    - Availability of Data: While labeled data can be expensive to obtain, unlabeled data can be cheap and plentiful. The promise of self-taught learning is that by exploiting the massive amount of unlabeled data, we can learn much better models.
        - By using unlabeled data to learn a good initial value for the weights in all the layers (except for the final layer that maps to the outputs/predictions), the algorithm is able to learn and discover patterns from massively more amounts of data than purely supervised approaches.
        - This often results in much better classifiers being learned.
    - Better Local Optima: After having trained the network on the unlabeled data, the weights are now starting at a better location in parameter space than if they had been randomly initialized. We can then further fine-tune the weights starting from this location.
        - Empirically, it turns out that gradient descent from this location is much more likely to lead to a good local minimum, because the unlabeled data has already provided a significant amount of "prior" information about what patterns there are in the input data.

SPARSE AUTOENCODER NOTATION

Symbol	Meaning
x	Input features for a training example, $x \in \Re^n$.
y	Output/target values. Here, y can be vector valued. In the case of an autoencoder, $y = x$.
$(x^{(i)}, y^{(i)})$	The i-th training example
$h_{W,b}(x)$	Predicted output for input x, using parameters W, b. For an autoencoder, it is the same as $\hat{x}$, else, it is $\hat{y}$.
$W_{ij}^{(l)}$	Synaptic weight parameter associated with the connection between unit j in layer l, and unit i in layer $l + 1$.
$b_i^{(l)}$	The bias term associated with unit i in layer $l + 1$.
θ	Parameter vector; result of taking all the parameters W, b and "unrolling" *them into a long column vector.*

$z_i^{(l)}$	Activation potential of unit i in layer l.
$f(\cdot)$	Activation function (e.g., $f(z) = \tanh(z)$).
$a_i^{(l)}$	Output (activation) of unit i in layer l of the network.
α	Learning rate parameter
s_l	Number of units in layer l (not counting the bias unit).
n_l	Number of layers in the network. L_1 is the input layer and layer L_{n_l} the output layer.
ρ	Sparsity parameter, which specifies our desired level of sparsity
$\hat{\rho}_i$	The average activation of hidden unit i (in the sparse autoencoder).
β	Weight of the sparsity penalty term (in the sparse autoencoder objective).

Stacked Autoencoders for Greedy Layer-Wise Training

- A stacked autoencoder is a neural network consisting of multiple layers of sparse autoencoders in which the outputs of each layer is wired to the inputs of the successive layer.
- Consider a stacked autoencoder with n layers. Let $W^{(k,1)}, W^{(k,2)}, b^{(k,1)}, b^{(k,2)}$ denote the parameters $W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)}$ for kth autoencoder.
- The encoding step for the stacked autoencoder is given by running the encoding step of each layer in forward order:

$$a^{(l)} = f(z^{(l)})$$

$$z^{(l+1)} = W^{(l,1)}a^{(l)} + b^{(l,1)}$$

- The decoding step is given by running the decoding stack of each autoencoder in reverse order:

$$a^{(n+l)} = f(z^{(n+l)})$$

$$z^{(n+l+1)} = W^{(n-l,2)}a^{(n+l)} + b^{(n-l,2)}$$

- The information of interest is contained within $a^{(n)}$, which is the activation of the deepest layer of hidden units. This vector gives a representation of the input in terms of higher-order features.
- The features from the stacked autoencoder can be used for the pattern recognition task (e.g., classification problems by feeding $a^{(n)}$ to a softmax classifier).

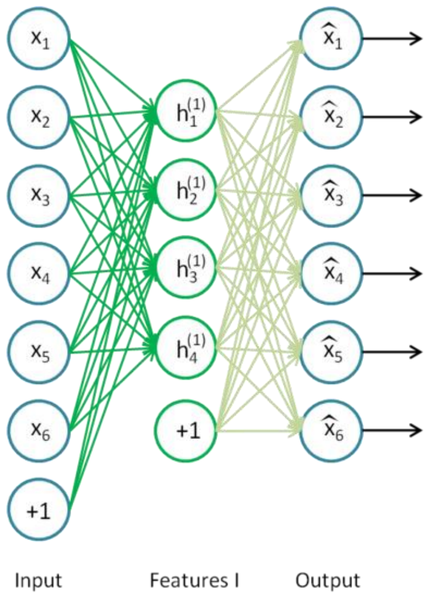
Greedy Layer-Wise Training

- First train the first layer on raw input to obtain parameters $W^{(1,1)}, W^{(1,2)}, b^{(1,1)}, b^{(1,2)}$. Use the first layer to transform the raw input into a vector consisting of activation of the hidden units, $a^{(1)}$. Train the second layer on this vector to obtain parameters $W^{(2,1)}, W^{(2,2)}, b^{(2,1)}, b^{(2,2)}$. Repeat for subsequent layers, using the output of each layer as input for the subsequent layer.
- This method trains the parameters of each layer individually while freezing parameters for the remainder of the model.

Example

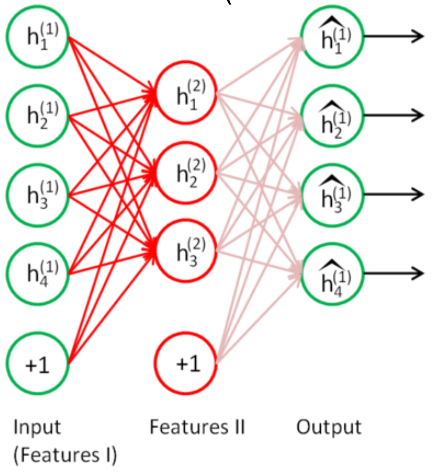
- To give a concrete example, suppose you wished to train a stacked autoencoder with 2 hidden layers for classification of MNIST digits, as we will do using the Matlab neural net toolbox.

- First, you would train a sparse autoencoder on the raw inputs $x^{(k)}$ to learn primary features $h^{(1)(k)}$ on the raw input.



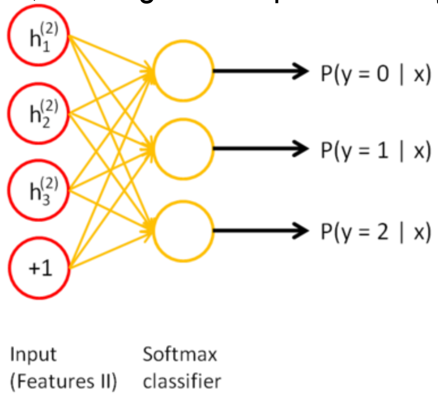
x_1 x_2 x_3 x_4 x_5 x_6 +1
$h_1^{(1)}$ $h_2^{(1)}$ $h_3^{(1)}$ $h_4^{(1)}$ +1
$\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$ $\hat{x}_4$ $\hat{x}_5$ $\hat{x}_6$
Input
Features I
Output

- Next, we feed the raw input into this trained sparse autoencoder, obtaining the primary feature activations $h^{(1)(k)}$ for each of the inputs $x^{(k)}$.
- Use these primary features (denoted “Features I”) as the "raw input" to another sparse autoencoder to learn secondary features $h^{(2)(k)}$ (denoted “Features II”).

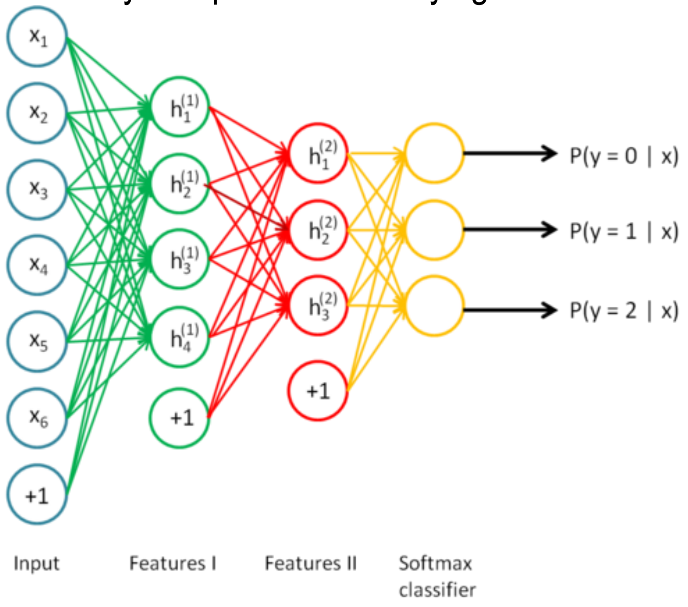


$h_1^{(1)}$ $h_2^{(1)}$ $h_3^{(1)}$ $h_4^{(1)}$ +1
$h_1^{(2)}$ $h_2^{(2)}$ $h_3^{(2)}$ +1
$\hat{h}_1^{(1)}$ $\hat{h}_2^{(1)}$ $\hat{h}_3^{(1)}$ $\hat{h}_4^{(1)}$
Input (Features I)
Features II
Output

- Feed the primary features into the second sparse autoencoder to obtain the secondary feature activations $h^{(2)(k)}$ for each of the primary features $h^{(1)(k)}$. We treat these secondary features as "raw input" to a softmax classifier, training it to map secondary features to digit labels.



- Finally, we combine all three layers together to form a stacked autoencoder with 2 hidden layers and a final softmax classifier layer capable of classifying the MNIST digits as desired.

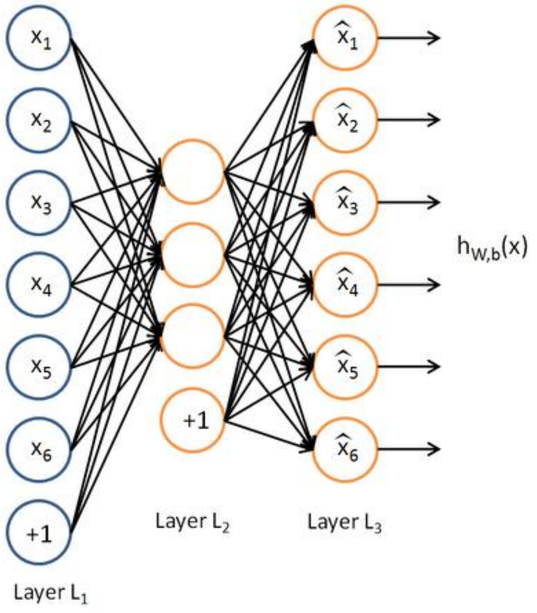


FINE-TUNING STACKED AEs

- Fine tuning is a strategy that is commonly found in deep learning. As such, it can also be used to greatly improve the performance of a stacked autoencoder.
- From a high level perspective, fine tuning treats all layers of a stacked autoencoder as a single model, so that in one iteration, we are improving upon all the weights in the stacked autoencoder.
- A backpropagation type of algorithm can be readily employed for fine tuning.
- Given that the network synaptic parameters are already tuned through autoencoding, network often converges to better local optima.

AUTOENCODERS AND SPARSITY

- Consider the following autoencoder trying to learn a function $h_{W,b}(x) \approx x$:



- The identity function seems a particularly trivial function to be trying to learn; but by placing constraints on the network, such as by limiting the number of hidden units, network is forced to learn a ***compressed representation*** of the input, yielding interesting structure about the data.
- **If the input were completely random**---say, each x_i comes from an IID Gaussian independent of the other features---**then this compression task would be very difficult**. But if there is structure in the data, for example, if some of the input features are correlated, then this algorithm will be able to discover some of those correlations.
- In fact, this simple autoencoder often ends up learning a low-dimensional representation very similar to PCAs.
- Our argument above relied on the number of hidden units s_2 being small. But even when the number of hidden units is large (perhaps even greater than the number of input pixels), we can still discover interesting structure, by imposing other constraints on the network.
- In particular, if we impose a "**sparsity**" constraint on the hidden units, then the autoencoder will still discover interesting structure in the data, even if the number of hidden units is large.
- Informally, we will think of a neuron as being "**active**" (or as "firing") **if its output value is close to 1**, or as being "inactive" if its output value is close to 0. This discussion assumes a sigmoid activation function. If you are using a tanh activation function, then we think of a neuron as being inactive when it outputs values close to -1.
- We would like to **constrain the neurons to be inactive most of the time**.
- Let $\hat{\rho}_j$ be the **average activation of hidden unit** j (averaged over the training set).

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}\left[a_j^{(2)}(x^{(i)})\right]$$

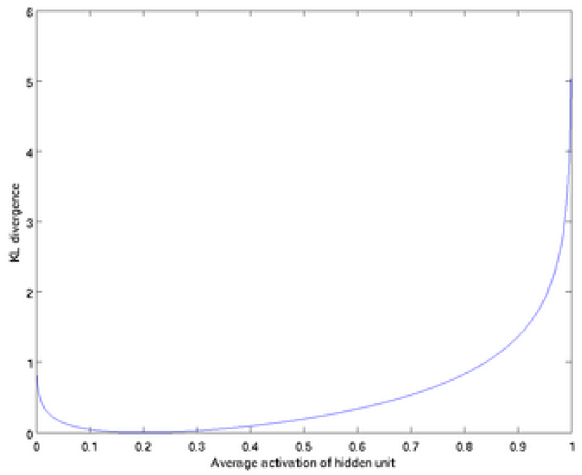
- We would like to (approximately) **enforce the constraint $\hat{\rho}_j = \rho$ where ρ is a "sparsity parameter"**, typically a **small value close to zero** (say $\rho = 0.05$).
- To satisfy this constraint, the hidden unit's activations must mostly be near 0.
- To achieve this, we will **add an extra penalty term to our optimization objective** that penalizes $\hat{\rho}_j$ deviating significantly from ρ. Many choices of the penalty term will give reasonable results.
- We will choose the following based on the concept of **Kullback-Leibler (KL) divergence**:

$$\sum_{j=1}^{s_2}\mathrm{KL}(\rho\|\hat{\rho}_j) = \sum_{j=1}^{s_2}\rho\log\frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j}.$$

- **KL divergence measures difference between a Bernoulli random variable[2] with mean ρ and a Bernoulli random variable with mean $\hat{\rho}_j$**; measures how different two different distributions are.
- Note: KL divergence is not symmetric!

[2] Bernoulli distribution is the probability distribution of a random variable which takes the value 1 with success probability of p and the value 0 with failure probability of $q = 1 - p$.

- This penalty function has the property that $KL(\rho || \hat{\rho}_j) = 0$ if $\hat{\rho}_j = \rho$, and otherwise it increases monotonically as $\hat{\rho}_j$ diverges from ρ. For example, in the figure below, we have set $\rho = 0.2$, and plotted $KL(\rho || \hat{\rho}_j)$ for a range of values of $\hat{\rho}_j$:



- We see that the KL-divergence reaches its minimum of 0 at $\hat{\rho}_j = \rho$, and blows up (it actually approaches ∞) as $\hat{\rho}_j$ approaches 0 or 1. Thus, minimizing this penalty term has the effect of causing $\hat{\rho}_j$ to be close to ρ.
- Our **overall cost function** is now

$$J_{\text{sparse}}(W, b) = J(W, b) + \beta \sum_{j=1}^{s_2} \text{KL}(\rho || \hat{\rho}_j),$$

  where $\boldsymbol{\beta}$ **controls the weight of the sparsity penalty term**.
- The term $\hat{\rho}_j$ (implicitly) depends on W, b also, because it is the average activation of hidden unit j, and the activation of a hidden unit depends on the parameters W, b.
- To incorporate the KL-divergence term into our derivative calculation, there is a simple-to-implement trick involving only a small change to the code. Specifically, where previously for the second layer ($l = 2$), during backpropagation we would have computed

$$\delta_i^{(2)} = \left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) f'(z_i^{(2)}),$$

  now instead we compute

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i} \right) \right) f'(z_i^{(2)}).$$

- <u>Need for Batch Training</u>: One subtlety is that you'll need to know $\hat{\rho}_i$ to compute this term. Thus, you'll need to compute a forward pass on all the training examples first to compute the average activations on the training set, before computing backpropagation on any example.

VISUALIZING A TRAINED AUTOENCODER

- Having trained a (sparse) autoencoder, we would now like to visualize the function learned by the algorithm, to try to understand what it has learned.

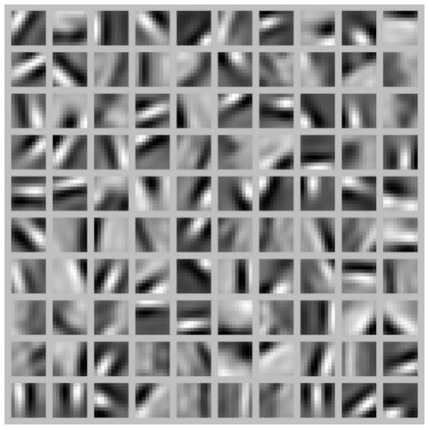
- Consider the case of training an autoencoder on 10×10 images, so that $n = 100$. Each hidden unit i computes a function of the input:

$$a_i^{(2)} = f\left(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_i^{(1)}\right).$$

- We will visualize the function computed by hidden unit i---which depends on the parameters $W_{ij}^{(1)}$ (ignoring the bias term for now)---using a 2D image.
- In particular, we think of $a_i^{(2)}$ as some non-linear feature of the input x. We ask: What input image x would cause $a_i^{(2)}$ to be maximally activated? (Less formally, what is the feature that hidden unit i is looking for?)
- For this question to have a non-trivial answer, we must impose some constraints on x.
- If we suppose that the input is norm constrained by $||x||^2 = \sum_{i=1}^{100} x_i^2 \leq 1$, then one can show that the input which maximally activates hidden unit i is given by setting pixel x_j (for all 100 pixels, $j = 1, \ldots, 100$) to

$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}}.$$

- By displaying the image formed by these pixel intensity values, we can begin to understand what feature hidden unit i is looking for.
- If we have an autoencoder with 100 hidden units (say), then we our visualization will have 100 such images---one per hidden unit. By examining these 100 images, we can try to understand what the ensemble of hidden units is learning.
- When we do this for a sparse autoencoder (trained with 100 hidden units on 10x10 pixel inputs[3] we get the following result:

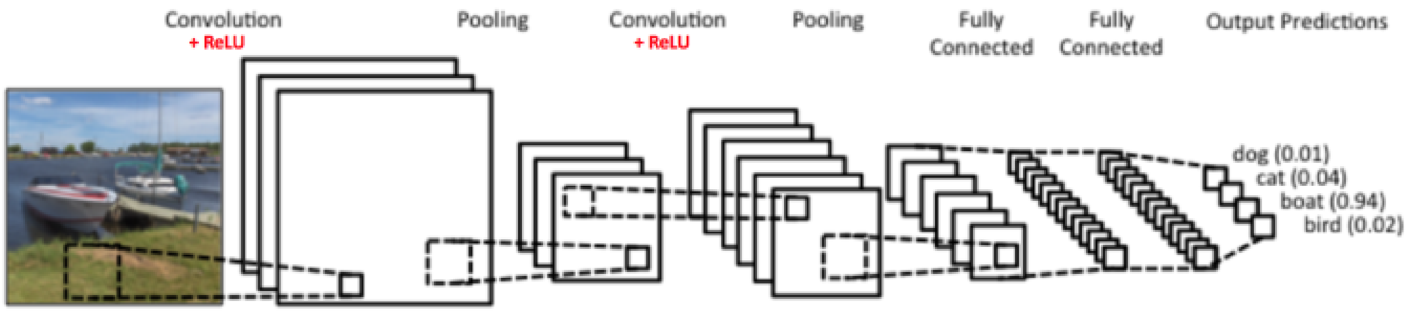


- Each square in the figure above shows the (norm bounded) input image x that maximally actives one of 100 hidden units. We see that the different hidden units have learned to detect edges at different positions and orientations in the image.
- These features are useful for such tasks as object recognition and other vision tasks. Similar useful representations are attained in other input domains as well (such as audio).

[3] The learned features were obtained by training on whitened natural images. Whitening is a preprocessing step which removes redundancy in the input, by causing adjacent pixels to become less correlated.

CONVOLUTION NEURAL NETWORKS (CNNS)[4]

- Convolutional neural networks (CNNs) are the current state-of-the-art model architecture for image classification tasks (as well as some text mining applications).
- CNNs apply a series of "filters" to the raw pixel data of an image to extract and learn higher-level features, which the model can then use for classification.
- CNNs contains three components:
  - **Convolutional layers**, which apply a specified number of convolution filters to the image. For each subregion, the layer performs a set of mathematical operations to produce a single value in the output feature map. Convolutional layers then typically apply a ReLU activation function to the output to introduce nonlinearities into the model.
  - **Pooling layers**, which downsample the image data extracted by the convolutional layers to reduce the dimensionality of the feature map in order to decrease processing time. A commonly used pooling algorithm is max pooling, which extracts subregions of the feature map (e.g., 2x2-pixel tiles), keeps their maximum value, and discards all other values.
  - **Dense (fully connected) layers**, which perform classification on the features extracted by the convolutional layers and down-sampled by the pooling layers. In a dense layer, every node in the layer is connected to every node in the preceding layer.



A simple CNN. [Source]

- Typically, a CNN is composed of a stack of convolutional modules that perform feature extraction. Each module consists of a convolutional layer followed by a pooling layer. The last convolutional module is followed by one or more dense layers that perform classification.
- The final dense layer in a CNN contains a single node for each target class in the model (all the possible classes the model may predict), with a softmax activation function to generate a value between 0–1 for each node (the sum of all these softmax values is equal to 1).
- We can interpret the softmax values for a given image as relative measurements of how likely it is that the image falls into each target class.

[4] Source: https://www.tensorflow.org/tutorials/layers

An Intuitive Explanation of Convolutional Neural Networks – Blog by Ujjwal Karn[5]

- Convolutional Neural Networks (CNNs) are a category of Neural Networks that have proven very effective in areas such as image recognition and classification.
- CNNs have been successful in identifying faces, objects and traffic signs apart from powering vision in robots and self driving cars.

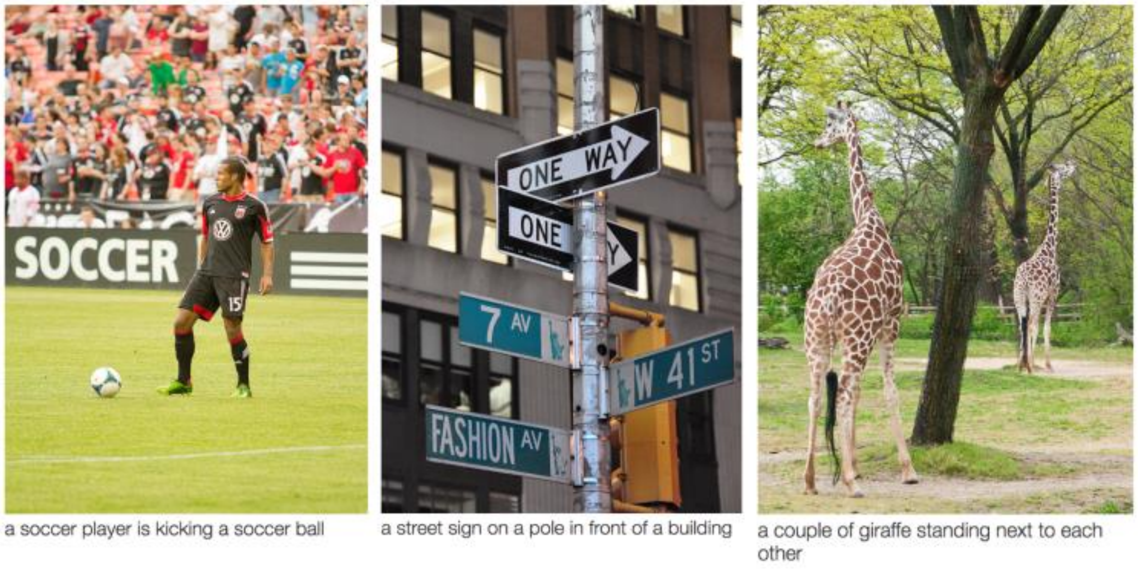

SOCCER
ONE WAY
ONE WAY
7 AV
W 41 ST
FASHION AV
a soccer player is kicking a soccer ball
a street sign on a pole in front of a building
a couple of giraffe standing next to each other

Figure 1: Source [1]

- In **Figure 1** above, a CNN is able to recognize scenes and the system is able to suggest relevant captions ("a soccer player is kicking a soccer ball") while **Figure 2** shows an example of CNNs being used for recognizing everyday objects, humans and animals.
- Lately, CNNs have been effective in several Natural Language Processing tasks (such as sentence classification) as well.

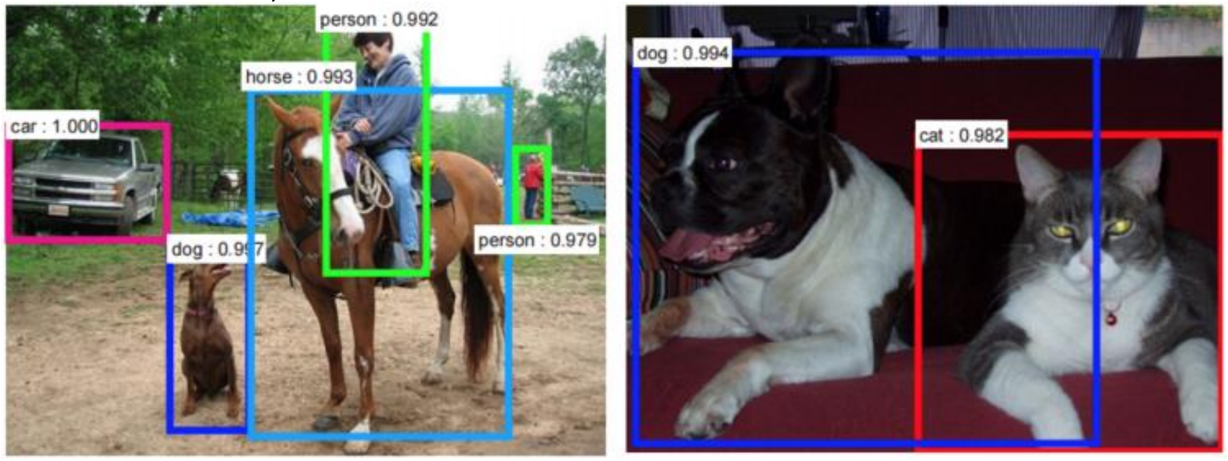


Figure 2: Source [2]

- Multi Layer Perceptrons are referred to as "Fully Connected Layers" in this post.

The LeNet Architecture (1990s)

- LeNet was one of the very first convolutional neural networks which helped propel the field of Deep Learning. This pioneering work by Yann LeCun was named LeNet5 [3].
- Below, we will develop an intuition of how the LeNet architecture learns to recognize images.

[5] Source: Blog by Ujjwal Karn- https://ujjwalkarn.me/2016/08/11/intuitive-explanation-CNNs/

- There have been several new architectures proposed in the recent years which are improvements over the LeNet, but they all use the main concepts from the LeNet and are relatively easier to understand if you have a clear understanding of the former.

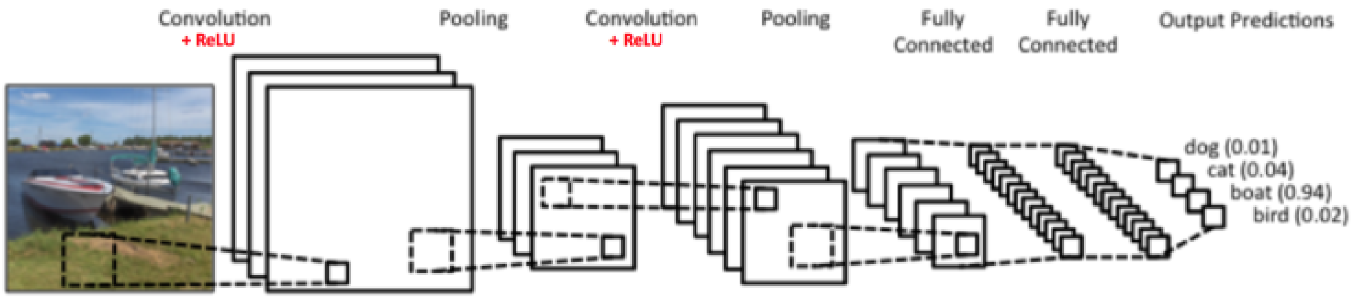


Figure 3: A simple CNN. Source [5]

- The Convolutional Neural Network in **Figure 3** is similar in architecture to the original LeNet and classifies an input image into four categories: dog, cat, boat or bird (the original LeNet was used mainly for character recognition tasks).
- As evident from the figure above, on receiving a boat image as input, the network correctly assigns the highest probability for boat (0.94) among all four categories. The sum of all probabilities in the output layer should be one (explained later in this post).
- There are four main operations in the CNN shown in **Figure 3** above:
  1. Convolution
  2. Non Linearity (ReLU)
  3. Pooling or Sub Sampling
  4. Classification (Fully Connected Layer)
- These operations are the basic building blocks of *every* Convolutional Neural Network, so understanding how these work is an important step to developing a sound understanding of CNNs. We will try to understand the intuition behind each of these operations below.

An Image is a Matrix of Pixel Values (3 Matrices for Color RGB Images: Red, Blue, Green)

- Essentially, every image can be represented as a matrix of pixel values.

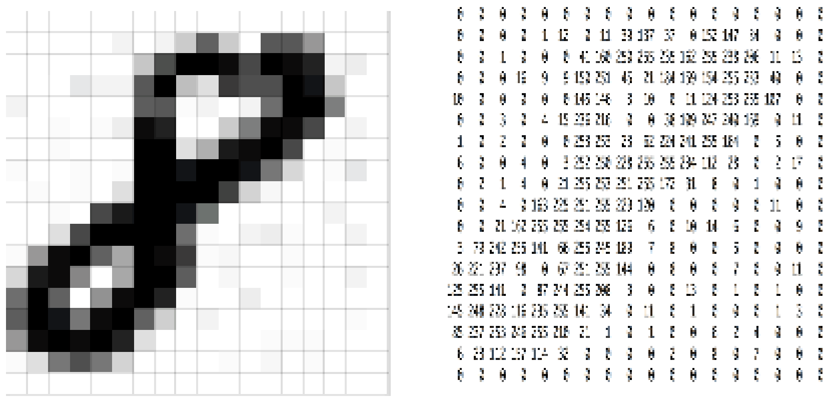


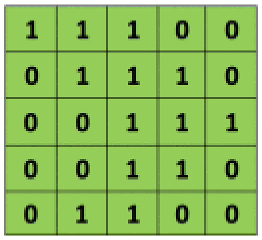
Figure 4: Every image is a matrix of pixel values. Source [6]

- Channel is a conventional term used to refer to a certain component of an image.
- An image from a standard digital camera will have three channels – red, green and blue – you can imagine those as three 2d-matrices stacked over each other (one for each color), each having pixel values in the range 0 to 255.

- A grayscale image, on the other hand, has just one channel. For the purpose of this post, we will only consider grayscale images, so we will have a single 2d matrix representing an image. The value of each pixel in the matrix will range from 0 to 255 – zero indicating black and 255 indicating white.

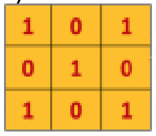
The Convolution Step

- The primary purpose of Convolution is to extract features from the input image.
- Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data.
- As we discussed above, every image can be considered as a matrix of pixel values.
- Consider a 5 x 5 image whose pixel values are only 0 and 1 (note that for a grayscale image, pixel values range from 0 to 255, the green matrix below is a special case where pixel values are only 0 and 1):



1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

- Also, consider another 3 x 3 matrix ("filter") as shown below:



1	0	1
0	1	0
1	0	1

- Then, the Convolution of the 5 x 5 image and the 3 x 3 matrix "filter" can be computed as shown in the below:

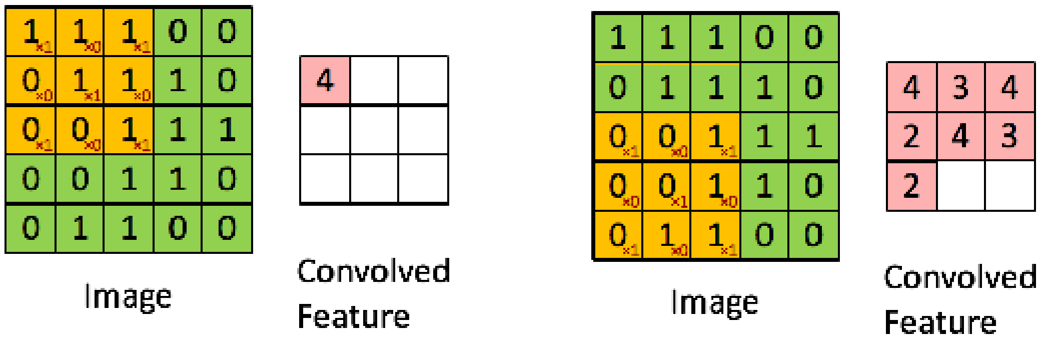
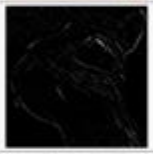
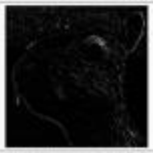
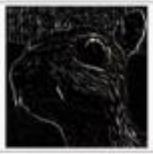
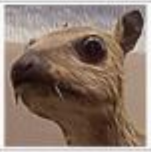
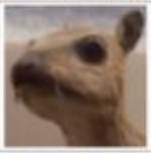
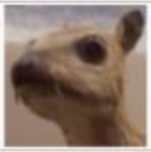


Figure 5: The Convolution operation. The output matrix is called Convolved Feature or Feature Map. Source [7]

- In the table below, we can see the effects of convolution of the above image with different filters. As shown, we can perform operations such as Edge Detection, Sharpen and Blur just by changing the numeric values of our filter matrix before the convolution operation [8] – this means that different filters can detect different features from an image, for example edges, curves etc. More such examples are available in Section 8.2.4 here.

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

- Another good way to understand the Convolution operation is by looking at the animation in **Figure 6** below:

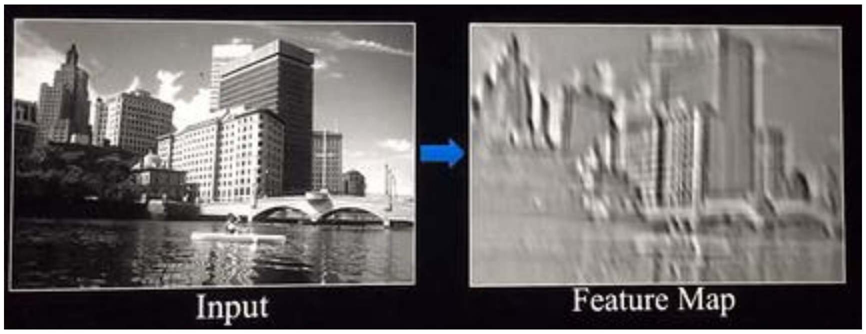


Figure 6: The Convolution Operation. Source [9]

- A filter (with red outline) slides over the input image (convolution operation) to produce a feature map. The convolution of another filter (with the green outline), over the same image gives a different feature map as shown. It is important to note that the Convolution operation captures the local dependencies in the original image. Also notice how these two different filters generate different feature maps from the same original image. Remember that the image and the two filters above are just numeric matrices as we have discussed above.

- In practice, a CNN *learns* the values of these filters on its own during the training process (although we still need to specify parameters such as number of filters, filter size, architecture of the network etc. before the training process). The more number of filters we have, the more image features get extracted and the better our network becomes at recognizing patterns in unseen images.
- The size of the Feature Map (Convolved Feature) is controlled by three parameters [4] that we need to decide before the convolution step is performed:
  - **Depth:** Depth corresponds to the number of filters we use for the convolution operation. In the network shown in **Figure 7**, we are performing convolution of the original boat image using three distinct filters, thus producing three different feature maps as shown. You can think of these three feature maps as stacked 2d matrices, so, the 'depth' of the feature map would be three.

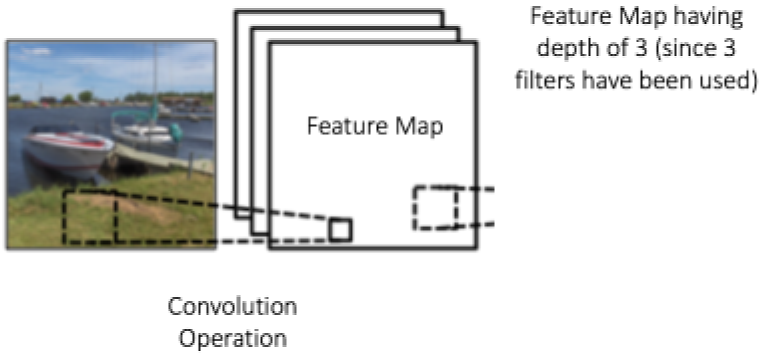

Feature Map having depth of 3 (since 3 filters have been used)
Feature Map
Convolution Operation

Figure 7: Feature Map with "Depth=3"

  - **Stride:** Stride is the number of pixels by which we slide our filter matrix over the input matrix. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2, then the filters jump 2 pixels at a time as we slide them around. Having a larger stride will produce smaller feature maps.
  - **Zero-padding:** Sometimes, it is convenient to pad the input matrix with zeros around the border, so that we can apply the filter to bordering elements of our input image matrix. A nice feature of zero padding is that it allows us to control the size of the feature maps. Adding zero-padding is also called *wide convolution*, and not using zero-padding would be a *narrow convolution*. This has been explained clearly in [14].

Introducing Non Linearity (ReLU)

- ReLU has been used after every Convolution operation in **Figure 3** above.
- ReLU stands for Rectified Linear Unit and is a non-linear operation. Its output is given by:

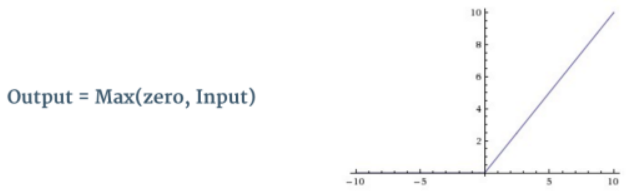


Figure 8: the ReLU operation

- ReLU is an element wise operation (applied per pixel) and replaces all negative pixel values in the feature map by zero. The purpose of ReLU is to introduce non-linearity in our CNN, since most of the real-world data we would want our CNN to learn would be non-linear (Convolution is a linear operation – element wise matrix multiplication and addition, so we account for non-linearity by introducing a non-linear function like ReLU).
- The ReLU operation can be understood clearly from **Figure 9** below. It shows the ReLU operation applied to one of the feature maps obtained in **Figure 6** above. The output feature map here is also referred to as the 'Rectified' feature map.

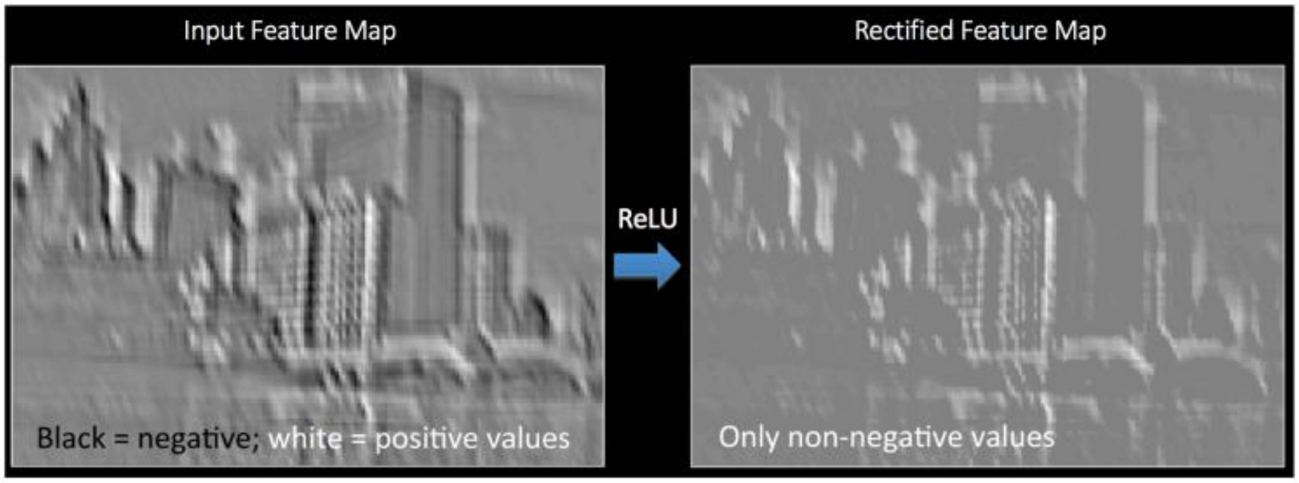


Figure 9: ReLU operation. Source [10]

- Other non linear functions such as **tanh** or **sigmoid** can also be used instead of ReLU, but ReLU has been found to perform better in most situations.

The Pooling Step

- Spatial Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map but retains the most important information. Spatial Pooling can be of different types: **Max**, **Average**, **Sum** etc.
- In case of Max Pooling, we define a spatial neighborhood (for example, a 2×2 window) and take the largest element from the rectified feature map within that window. Instead of taking the largest element we could also take the average (Average Pooling) or sum of all elements in that window. In practice, Max Pooling has been shown to work better.
- **Figure 10** shows an example of Max Pooling operation on a Rectified Feature map (obtained after convolution + ReLU operation) by using a 2×2 window.

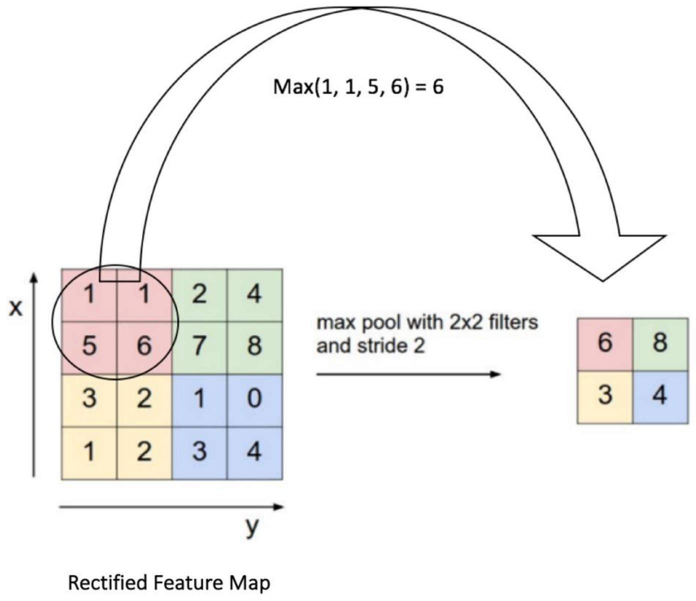


Figure 10: Max Pooling. Source [4]

- We slide our 2 x 2 window by 2 cells (also called 'stride') and take the maximum value in each region. As shown in **Figure 10**, this reduces the dimensionality of our feature map.
- In the network shown in **Figure 11,** pooling operation is applied separately to each feature map (notice that, due to this, we get three output maps from three input maps).

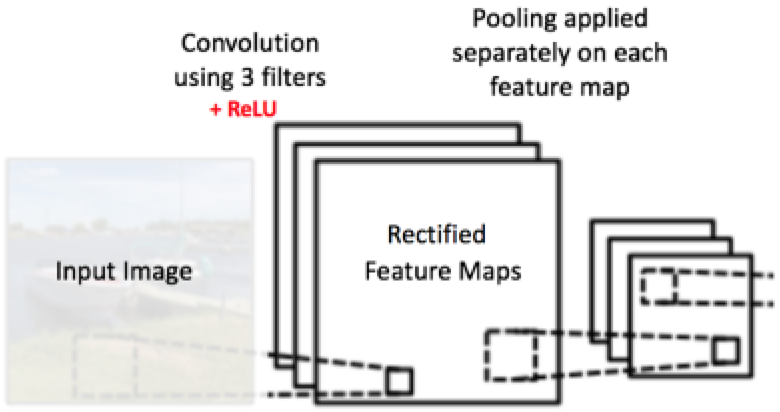

Convolution using 3 filters
+ ReLU
Pooling applied separately on each feature map
Input Image
Rectified Feature Maps

Figure 11: Pooling applied to Rectified Feature Maps

- **Figure 12** shows the effect of Pooling on the Rectified Feature Map we received after the ReLU operation in **Figure 9** above.

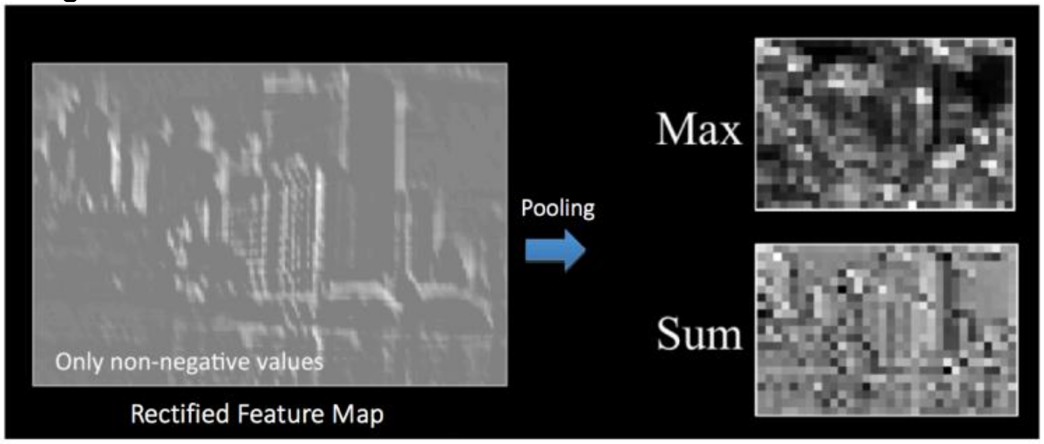


Figure 12: Pooling. Source [10]

- The function of Pooling is to progressively reduce the spatial size of the input representation [4]. In particular, pooling
  - makes the input representations (feature dimension) smaller and more manageable
  - reduces the number of parameters and computations in the network, therefore, controlling overfitting [4]
  - makes the network invariant to small transformations, distortions and translations in the input image (a small distortion in input will not change the output of Pooling – since we take the maximum / average value in a local neighborhood).
  - helps us arrive at an almost scale invariant representation of our image (the exact term is "equivariant"). This is very powerful since we can detect objects in an image no matter where they are located (read [18] and [19] for details).

Story So Far

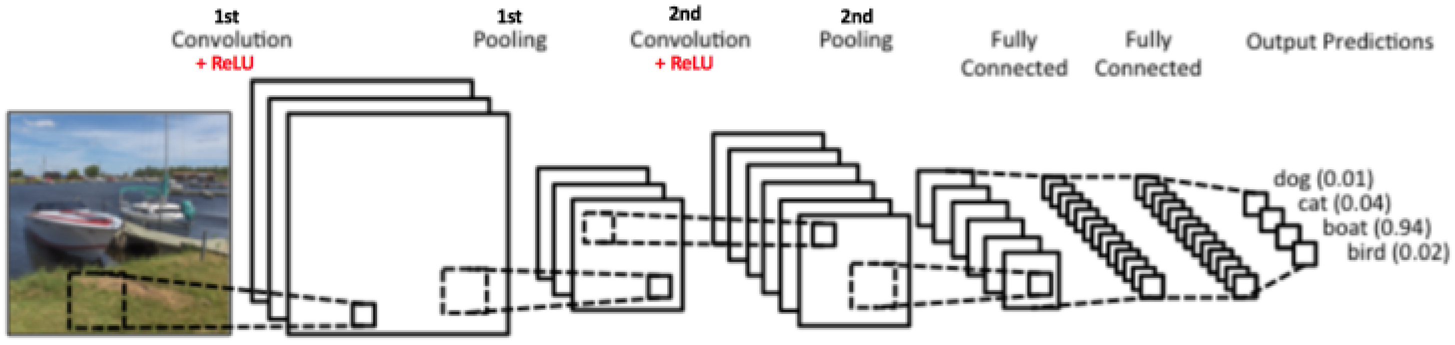


Figure 13: Simple CNN

- So far we have seen how Convolution, ReLU and Pooling work. It is important to understand that these layers are the basic building blocks of any CNN. As shown in **Figure 13**, we have two sets of Convolution, ReLU & Pooling layers – the 2nd Convolution layer performs convolution on the output of the first Pooling Layer using six filters to produce a total of six feature maps. ReLU is then applied individually on all of these six feature maps. We then perform Max Pooling operation separately on each of the six rectified feature maps.
- Together these layers extract the useful features from the images, introduce non-linearity in our network and reduce feature dimension while aiming to make the features somewhat equivariant to scale and translation [18].
- The output of the 2nd Pooling Layer acts as an input to the Fully Connected Layer, which we will discuss in the next section.

Fully Connected Layer

- The Fully Connected layer is a traditional Multi Layer Perceptron that uses a softmax activation function in the output layer (other classifiers like SVM can also be used, but will stick to softmax in this post). The term "Fully Connected" implies that every neuron in the previous layer is connected to every neuron on the next layer. I recommend reading this post if you are unfamiliar with Multi Layer Perceptrons.
- The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of the Fully Connected layer is to use these features for classifying the

input image into various classes based on the training dataset. For example, the image classification task we set out to perform has four possible outputs as shown in **Figure 14** below (note that Figure 14 does not show connections between the nodes in the fully connected layer)

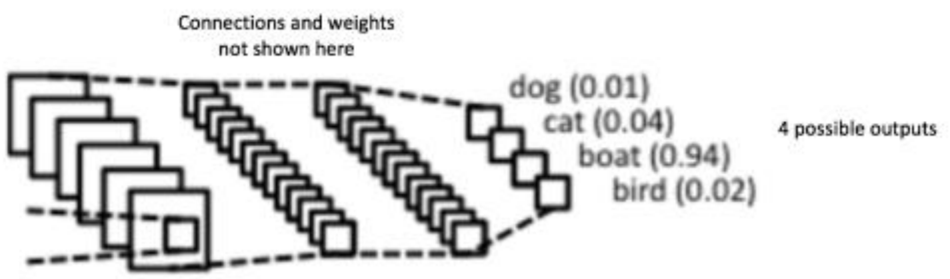


Figure 14: Fully Connected Layer -each node is connected to every other node in the adjacent layer

- Apart from classification, adding a fully-connected layer is also a (usually) cheap way of learning non-linear combinations of these features. Most of the features from convolutional and pooling layers may be good for the classification task, but combinations of those features might be even better [11].
- The sum of output probabilities from the Fully Connected Layer is 1. This is ensured by using the Softmax as the activation function in the output layer of the Fully Connected Layer. The Softmax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that sum to one.

Putting it all together – Training using Backpropagation

- As discussed above, the Convolution + Pooling layers act as Feature Extractors from the input image while Fully Connected layer acts as a classifier.
- Note that in **Figure 15** below, since the input image is a boat, the target probability is 1 for Boat class and 0 for other three classes, i.e.
  - Input Image = Boat
  - Target Vector = [0, 0, 1, 0]

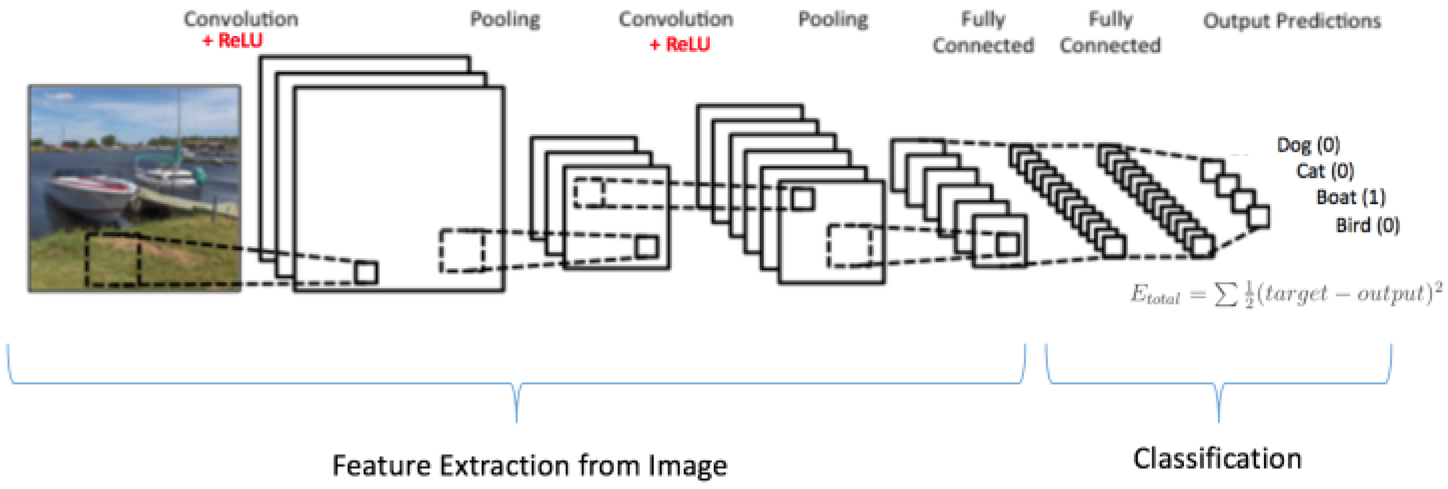


Figure 15: Training the CNN

- The overall training process of the Convolution Network may be summarized as below:

- **Step1:** We initialize all filters and parameters / weights with random values
- **Step2:** The network takes a training image as input, goes through the forward propagation step (convolution, ReLU and pooling operations along with forward propagation in the Fully Connected layer) and finds the output probabilities for each class.
    - Lets say the output probabilities for the boat image above are [0.2, 0.4, 0.1, 0.3]
    - Since weights are randomly assigned for the first training example, output probabilities are also random.
- **Step3:** Calculate the total error at the output layer (summation over all 4 classes)
- **Step4:** Use Backpropagation to calculate the *gradients* of the error with respect to all weights in the network and use *gradient descent* to update all filter values / weights and parameter values to minimize the output error.
    - The weights are adjusted in proportion to their contribution to the total error.
    - When the same image is input again, output probabilities might now be [0.1, 0.1, 0.7, 0.1], which is closer to the target vector [0, 0, 1, 0].
    - This means that the network has *learnt* to classify this particular image correctly by adjusting its weights / filters such that the output error is reduced.
    - Parameters like number of filters, filter sizes, architecture of the network etc. have all been fixed before Step 1 and do not change during training process – only the values of the filter matrix and connection weights get updated.
- **Step5:** Repeat steps 2-4 with all images in the training set.

- The above steps *train* the CNN – this essentially means that all the weights and parameters of the CNN have now been optimized to correctly classify images from the training set.
- When a new (unseen) image is input into the CNN, the network would go through the forward propagation step and output a probability for each class (for a new image, the output probabilities are calculated using the weights which have been optimized to correctly classify all the previous training examples). If our training set is large enough, the network will (hopefully) generalize well to new images and classify them into correct categories.
- **Note 1:** The steps above have been oversimplified and mathematical details have been avoided to provide intuition into the training process. See [4] and [12] for a mathematical formulation and thorough understanding.
- **Note 2:** In the example above we used two sets of alternating Convolution and Pooling layers. Please note however, that these operations can be repeated any number of times in a single CNN. In fact, some of the best performing CNNs today have tens of Convolution and Pooling layers! Also, it is not necessary to have a Pooling layer after every Convolutional Layer. As can be seen in the **Figure 16** below, we can have multiple Convolution + ReLU operations in succession before having a Pooling operation. Also notice how each layer of the CNN is visualized in the Figure 16 below.

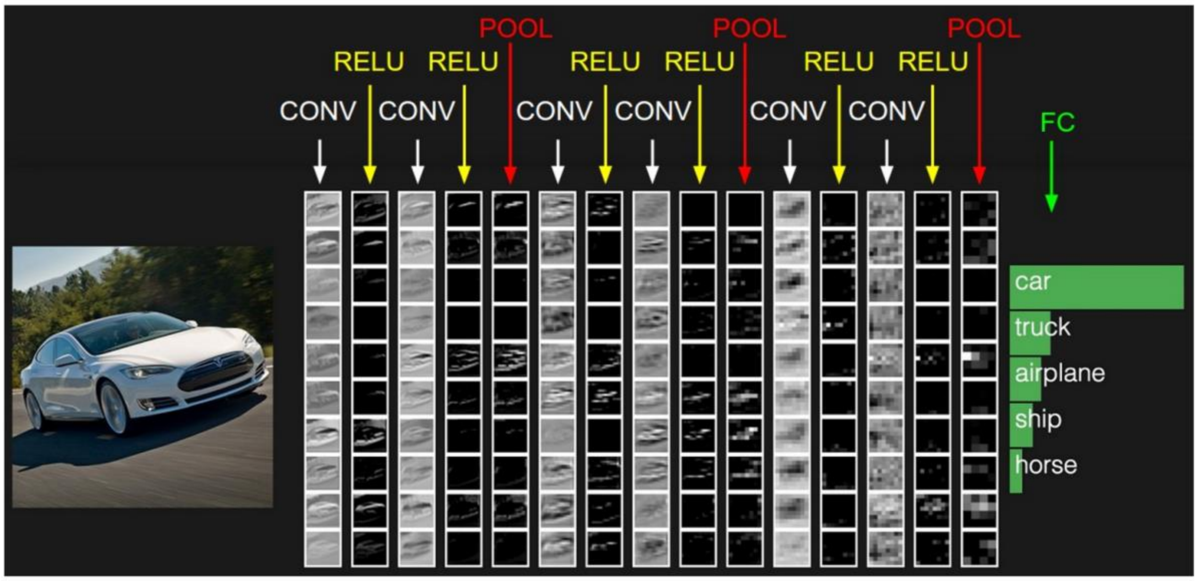


Figure 16: Source [4]

Visualizing Convolutional Neural Networks

- In general, the more convolution steps we have, the more complicated features our network will be able to learn to recognize. For example, in Image Classification a CNN may learn to detect edges from raw pixels in the first layer, then use the edges to detect simple shapes in the second layer, and then use these shapes to deter higher-level features, such as facial shapes in higher layers [14]. This is demonstrated in **Figure 17** below – these features were learnt using a Convolutional Deep Belief Network and the figure is included here just for demonstrating the idea (this is only an example: real life convolution filters may detect objects that have no meaning to humans).

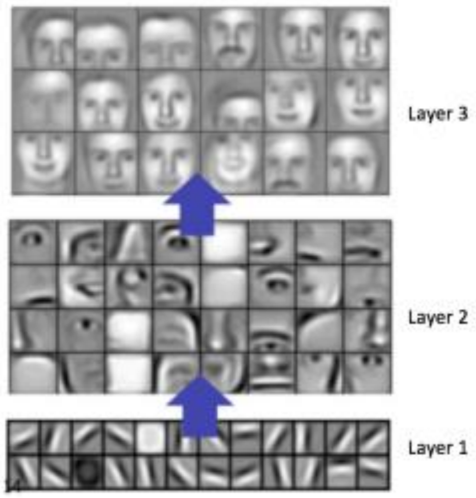


Figure 17: Learned features from a Convolutional Deep Belief Network. Source [21]

- Adam Harley created amazing visualizations of a Convolutional Neural Network trained on the MNIST Database of handwritten digits [13]. I highly recommend playing around with it to understand details of how a CNN works.
- We will see below how the network works for an input '8'. Note that the visualization in **Figure 18** does not show the ReLU operation separately.

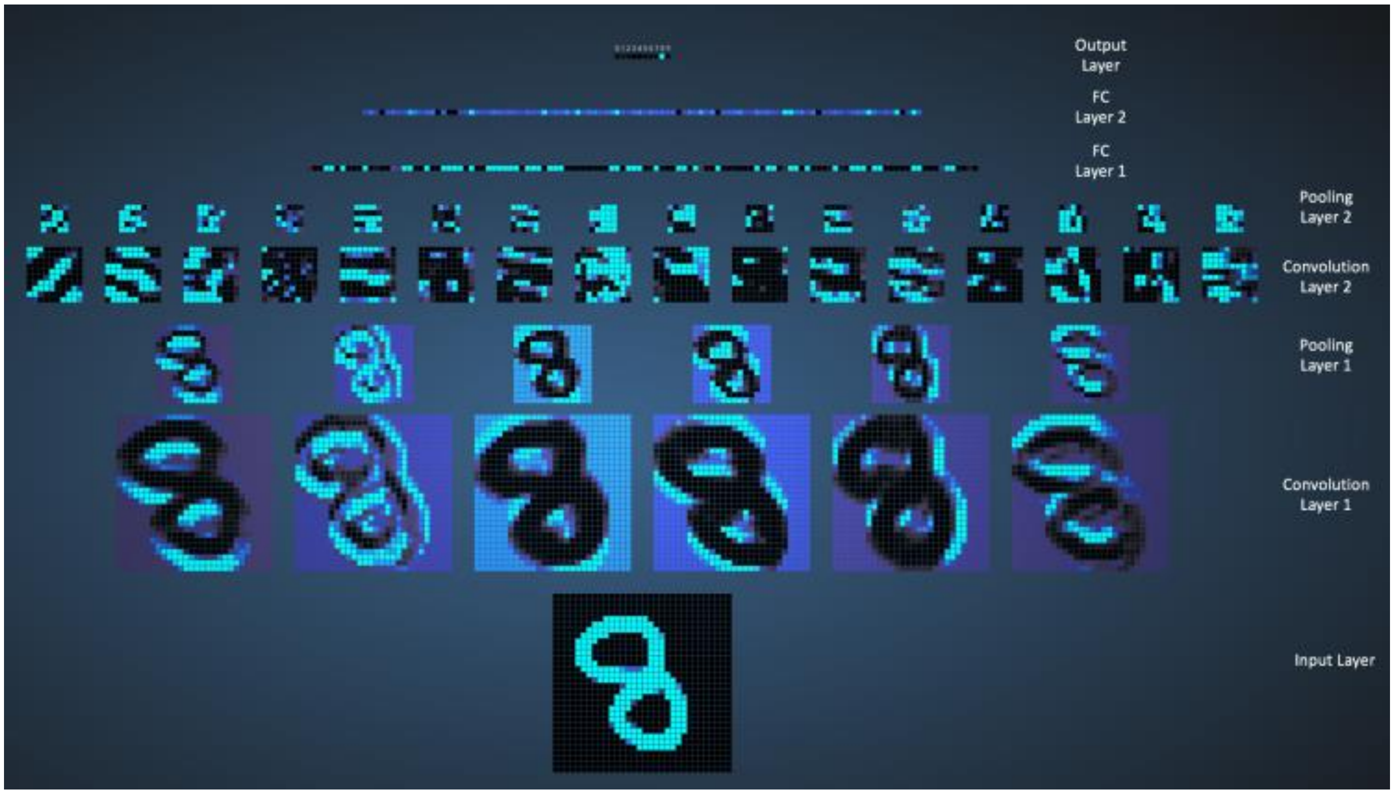


Figure 18: Visualizing a CNN trained on handwritten digits. Source [13]

- The input image contains 1024 pixels (32 x 32 image) and the first Convolution layer (Convolution Layer 1) is formed by convolution of six unique 5 × 5 (stride 1) filters with the input image. As seen, using six different filters produces a feature map of depth six.
- Convolutional Layer 1 is followed by Pooling Layer 1 that does 2 × 2 max pooling (with stride 2) separately over the six feature maps in Convolution Layer 1. You can move your mouse pointer over any pixel in the Pooling Layer and observe the 2 x 2 grid it forms in the previous Convolution Layer (demonstrated in **Figure 19**). You'll notice that the pixel having the maximum value (the brightest one) in the 2 x 2 grid makes it to the Pooling layer.

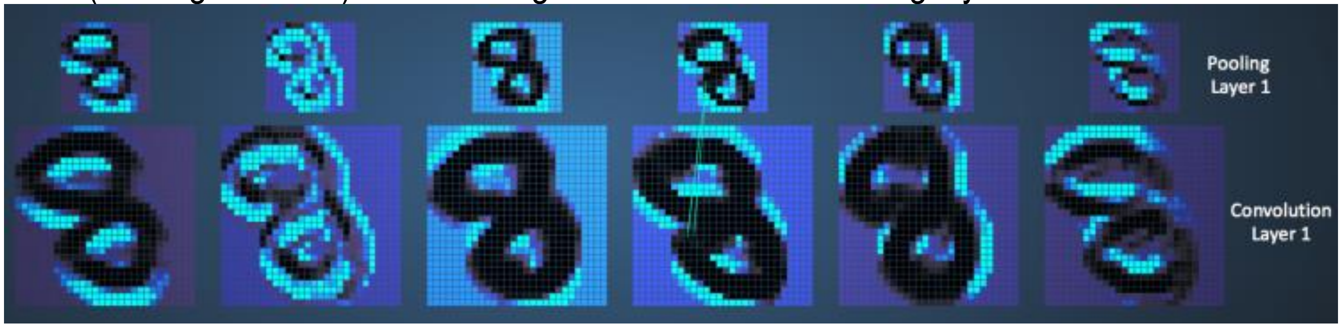


Figure 19: Visualizing the Pooling Operation. Source [13]

- Pooling Layer 1 is followed by sixteen 5 × 5 (stride 1) convolutional filters that perform the convolution operation. This is followed by Pooling Layer 2 that does 2 × 2 max pooling (with stride 2). These two layers use the same concepts as described above.
- We then have three fully-connected (FC) layers. There are:
  - 120 neurons in the first FC layer
  - 100 neurons in the second FC layer
  - 10 neurons in the third FC layer corresponding to the 10 digits – also called the Output layer

- Notice how in **Figure 20**, each of the 10 nodes in the output layer are connected to all 100 nodes in the 2nd Fully Connected layer (hence the name Fully Connected).
- Also, note how the only bright node in the Output Layer corresponds to '8' – this means that the network correctly classifies our handwritten digit (brighter node denotes that the output from it is higher, i.e. 8 has the highest probability among all other digits).

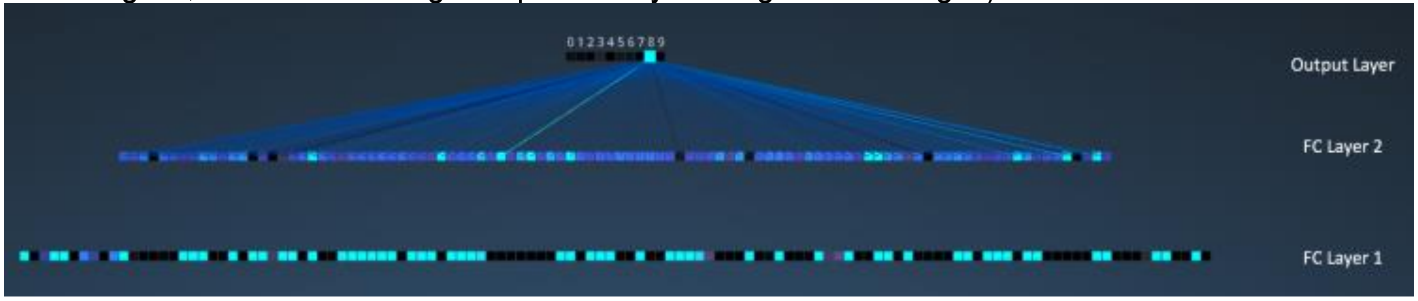


Figure 20: Visualizing the Filly Connected Layers. Source [13]

- The 3d version of the same visualization is available here.

Other CNN Architectures

- Convolutional Neural Networks have been around since early 1990s. We discussed the LeNet above which was one of the very first convolutional neural networks. Some other influential architectures are listed below [3] [4].
    - **LeNet (1990s):** Already covered in this article.
    - **1990s to 2012:** In the years from late 1990s to early 2010s convolutional neural network were in incubation. As more and more data and computing power became available, tasks that convolutional neural networks could tackle became more and more interesting.
    - **AlexNet (2012) –** In 2012, Alex Krizhevsky (and others) released AlexNet which was a deeper and much wider version of the LeNet and won by a large margin the difficult ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. It was a significant breakthrough with respect to the previous approaches and the current widespread application of CNNs can be attributed to this work.
    - **ZF Net (2013) –** The ILSVRC 2013 winner was a Convolutional Network from Matthew Zeiler and Rob Fergus. It became known as the ZFNet (short for Zeiler & Fergus Net). It was an improvement on AlexNet by tweaking the architecture hyperparameters.
    - **GoogLeNet (2014) –** The ILSVRC 2014 winner was a Convolutional Network from Szegedy et al.from Google. Its main contribution was the development of an *Inception Module* that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M).
    - **VGGNet (2014) –** The runner-up in ILSVRC 2014 was the network that became known as the VGGNet. Its main contribution was in showing that the depth of the network (number of layers) is a critical component for good performance.
    - **ResNets (2015) –** Residual Network developed by Kaiming He (and others) was the winner of ILSVRC 2015. ResNets are currently by far state of the art Convolutional Neural Network models and are the default choice for using CNNs in practice (as of May 2016).

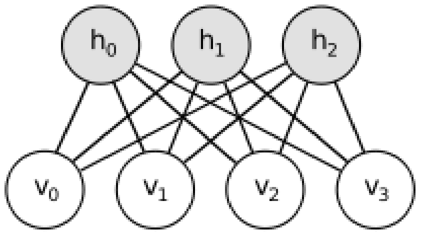
- **DenseNet (August 2016) –** Recently published by Gao Huang (and others), the Densely Connected Convolutional Network has each layer directly connected to every other layer in a feed-forward fashion. The DenseNet has been shown to obtain significant improvements over previous state-of-the-art architectures on five highly competitive object recognition benchmark tasks. Check out the Torch implementation here.

References

1. karpathy/neuraltalk2: Efficient Image Captioning code in Torch, Examples
2. Shaoqing Ren, *et al,* "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", 2015, arXiv:1506.01497
3. Neural Network Architectures, Eugenio Culurciello's blog
4. CS231n Convolutional Neural Networks for Visual Recognition, Stanford
5. Clarifai / Technology
6. Machine Learning is Fun! Part 3: Deep Learning and Convolutional Neural Networks
7. Feature extraction using convolution, Stanford
8. Wikipedia article on Kernel (image processing)
9. Deep Learning Methods for Vision, CVPR 2012 Tutorial
10. Neural Networks by Rob Fergus, Machine Learning Summer School 2015
11. What do the fully connected layers do in CNNs?
12. Convolutional Neural Networks, Andrew Gibiansky
13. A. W. Harley, "An Interactive Node-Link Visualization of Convolutional Neural Networks," in ISVC, pages 867-877, 2015 (link). Demo
14. Understanding Convolutional Neural Networks for NLP
15. Backpropagation in Convolutional Neural Networks
16. A Beginner's Guide To Understanding Convolutional Neural Networks
17. Vincent Dumoulin, *et al*, "A guide to convolution arithmetic for deep learning", 2015, arXiv:1603.07285
18. What is the difference between deep learning and usual machine learning?
19. How is a convolutional neural network able to learn invariant features?
20. A Taxonomy of Deep Convolutional Neural Nets for Computer Vision
21. Honglak Lee, *et al*, "Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations" (link)

Restricted Boltzmann Machines for Greedy Layer-Wise Training

- Instead of autoencoders, one can also employ Boltzmann Machines (BMs) for building deep neural networks.
- Boltzmann Machines (BMs) are a particular form of Markov Random Field (MRF), i.e., for which the energy function is linear in its free parameters.
- To make them powerful enough to represent complicated distributions (i.e., go from the limited parametric setting to a non-parametric one), we consider that some of the variables are never observed (they are called hidden).
- By having more hidden variables (also called hidden units), we can increase the modeling capacity of the Boltzmann Machine (BM).
- Restricted Boltzmann Machines further restrict BMs to those without visible-visible and hidden-hidden connections.



Graphical Depiction of an RBM

- Restricted Boltzmann machines (RBMs) are probabilistic graphical models that can be interpreted as stochastic neural networks (that is a network of neurons where each neuron have some random behavior when activated).
- They attracted much attention recently after being proposed as building blocks of multi-layer learning systems called ***deep belief networks***.
- Connections between neurons are bidirectional and symmetric; information flows in both directions during the training and usage and the weights are the same in both directions.
- The energy function $E(v,h)$ of an RBM is defined as:
$$E(v,h) = -b'v - c'h - h'Wv$$
where W represents the weights connecting hidden and visible units and b, c are the offsets of the visible and hidden layers respectively.
- This translates directly to the following free energy formula:
$$\mathcal{F}(v) = -b'v - \sum_i \log \sum_{h_i} e^{h_i(c_i + W_i v)}.$$
- Because of the specific structure of RBMs, visible and hidden units are conditionally independent given one-another:
$$p(h|v) = \prod_i p(h_i|v)$$
$$p(v|h) = \prod_j p(v_j|h).$$

RBMs with Binary Units

- In the commonly studied case of using binary units (where v_j and $h_i \in \{0,1\}$), we obtain a probabilistic version of the usual neuron activation function:
$$P(h_i = 1|v) = sigm(c_i + W_i v)$$

$$P(v_j = 1|h) = sigm(b_j + W'_j h)$$

- The free energy of an RBM with binary units further simplifies to:

$$\mathcal{F}(v) = -b'v - \sum_i \log(1 + e^{(c_i + W_i v)}).$$

Update Equations with Binary Units

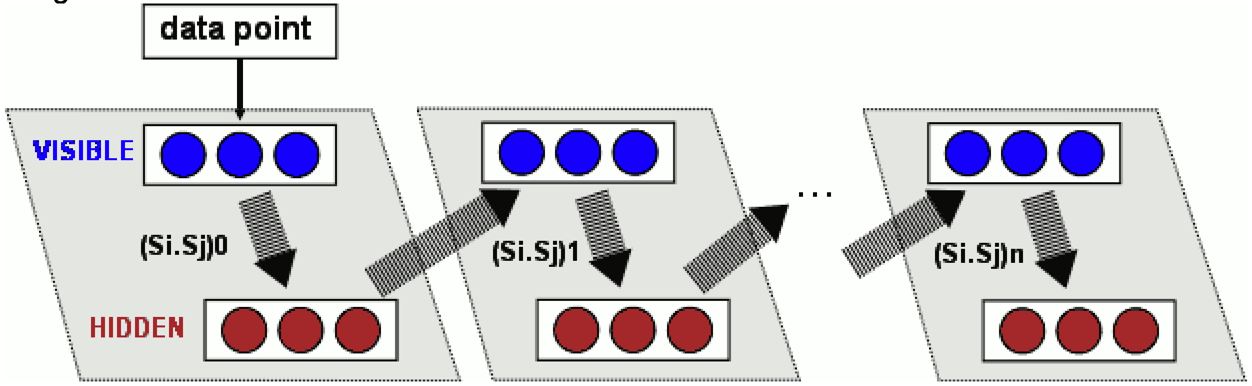
- Combining the above equations, we obtain the following log-likelihood gradients for an RBM with binary units:

$$-\frac{\partial \log p(v)}{\partial W_{ij}} = E_v[p(h_i|v) \cdot v_j] - v_j^{(i)} \cdot sigm(W_i \cdot v^{(i)} + c_i)$$

$$-\frac{\partial \log p(v)}{\partial c_i} = E_v[p(h_i|v)] - sigm(W_i \cdot v^{(i)})$$

$$-\frac{\partial \log p(v)}{\partial b_j} = E_v[p(v_j|h)] - v_j^{(i)}$$

- Both learning and usage require multiple cycles of forward/backward calculations till convergence is achieved.



Training and Usage of Restricted Boltzmann Machine

- For more information, read the following article: Fischer, A., & Igel, C. (2012, September). An introduction to restricted Boltzmann machines. *In Iberoamerican Congress on Pattern Recognition* (pp. 14-36). Springer Berlin Heidelberg. [LINK]

RECTIFIED LINEAR UNITS FOR DEEP NETWORKS

- In the context of artificial neural networks, the ***rectifier*** is an activation function defined as

$$f(x) = \max(0, x)$$

  where x is the input to a neuron.
- Also known as a ramp function, it is first introduced to a dynamical network by Hahnloser et al in a 2000 paper in Nature[1] with strong biological motivations and mathematical justifications.
- As of 2015, it is the most popular activation function for convolutional networks (more than the logistic sigmoid or the hyperbolic tangent).
- A unit employing the rectifier is also called a ***rectified linear unit*** (ReLU).
- A smooth approximation to the rectifier is the analytic function

$$f(x) = \ln(1 + e^x)$$

  which is called the ***softplus function***.
- The derivative of softplus is the standard logistic function.

IMPLEMENTING DEEP NEURAL NETWORKS USING MATLAB NEURAL NET TOOLBOX [LINK]

- Matlab allows construction and training of convolutional neural networks (CNNs, CNNs) for classification and autoencoder neural networks for learning features.
- **Convolutional Neural Networks**: Especially suited for image recognition, they allow classification, feature extraction, and transfer learning using convolutional neural networks
  - Functions: Construct Network Architecture

imageInputLayer	Image input layer
convolution2dLayer	Convolutional layer
reluLayer	Rectified Linear Unit (ReLU) layer
crossChannelNormalizationLayer	Channel-wise local response normalization layer
averagePooling2dLayer	Average pooling layer object
maxPooling2dLayer	Max pooling layer
fullyConnectedLayer	Fully connected layer
dropoutLayer	Dropout layer
softmaxLayer	Softmax layer
classificationLayer	Create a classification output layer

  - Functions: Train Network

trainingOptions	Options for training neural network
trainNetwork	Train a network

  - Functions: Extract Features and Predict Outcomes

activations	Compute network layer activations
predict	Predict responses using a trained network
classify	Classify data using a trained network

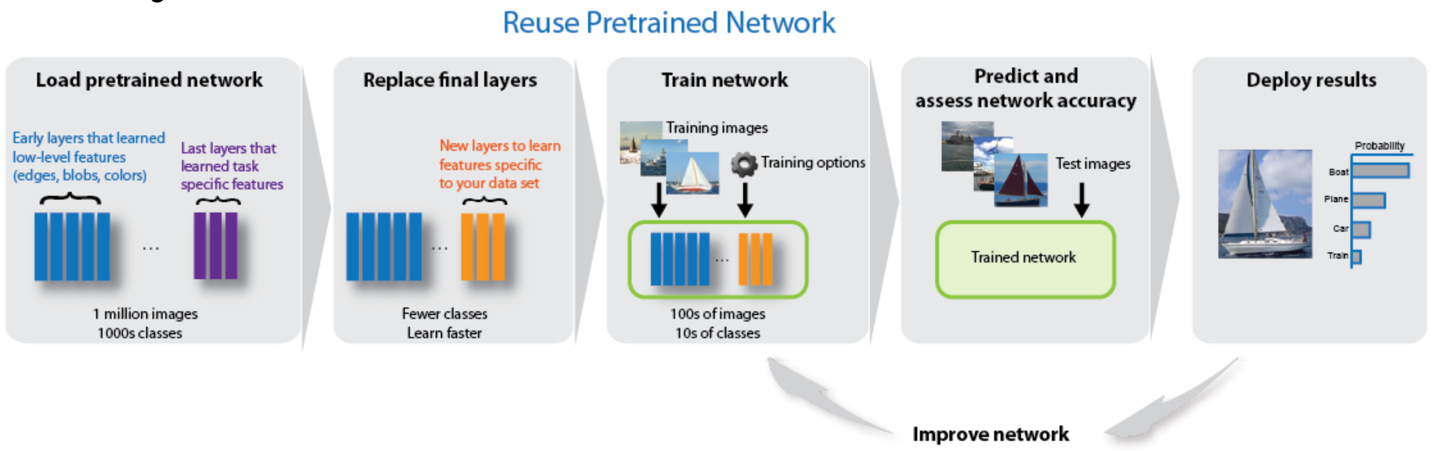
- **Autoencoders**: Perform unsupervised learning of features using autoencoder neural networks.
  - Functions:

trainAutoencoder	Train an autoencoder
trainSoftmaxLayer	Train a softmax layer for classification
decode	Decode encoded data
encode	Encode input data
generateFunction	Generate a MATLAB function to run the autoencoder
generateSimulink	Generate a Simulink model for the autoencoder
network	Convert Autoencoder object into network object
plotWeights	Plot a visualization of the weights for the encoder of an autoencoder
predict	Reconstruct the inputs using trained autoencoder
stack	Stack encoders from several autoencoders together
view	View autoencoder

- **Lot more functionality available recently: Check documentation!**

Matlab: Transfer Learning Using AlexNet [Source]

- This example shows how to fine-tune a pretrained AlexNet convolutional neural network to perform classification on a new collection of images.
- AlexNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals).
  - The network has learned rich feature representations for a wide range of images.
  - The network takes an image as input and outputs a label for the object in the image together with the probabilities for each of the object categories.
- Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task.
  - Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch.
  - You can quickly transfer learned features to a new task using a smaller number of training images.

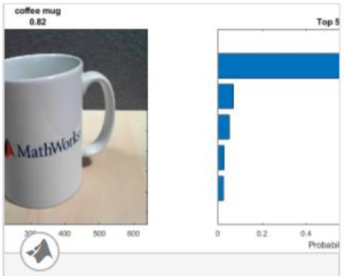


Matlab: Deep Learning Examples and Use Cases [Source]

Deep Learning Basics:

Classify Image Using GoogLeNet

Classify an image using the pretrained deep convolutional neural network GoogLeNet.

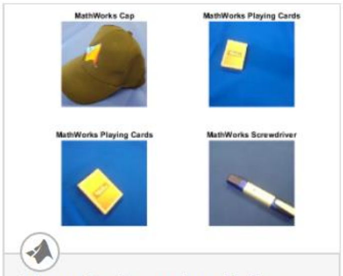


Classify Webcam Images Using Deep Learning

Classify images from a webcam in real time using the pretrained deep convolutional neural network AlexNet.

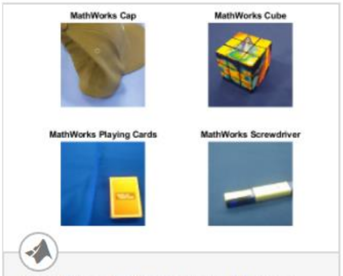
Create Simple Deep Learning Network for Classification

Create and train a simple convolutional neural network for deep learning classification. Convolutional neural networks are



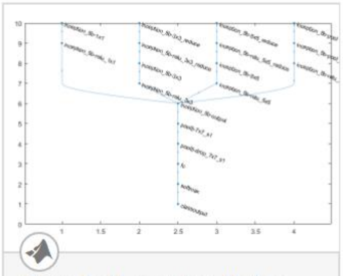
Transfer Learning Using AlexNet

Fine-tune a pretrained AlexNet convolutional neural network to perform classification on a new collection of images.



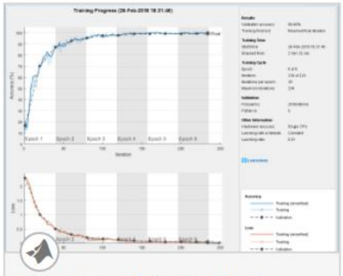
Feature Extraction Using AlexNet

Extract learned image features from a pretrained convolutional neural network, and use those features to train an image classifier. Feature



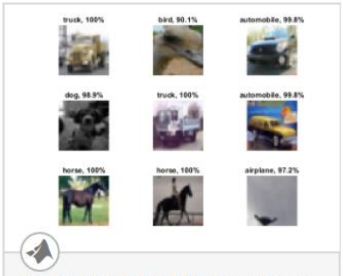
Transfer Learning Using GoogLeNet

Use transfer learning to retrain GoogLeNet, a pretrained convolutional neural network, to classify a new set of images.



Create and Train DAG Network for Deep Learning

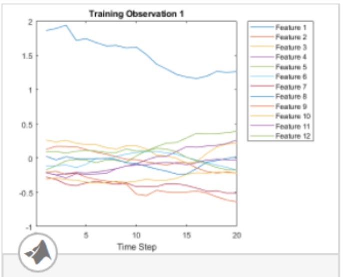
Create a simple directed acyclic graph (DAG) network for deep learning. Train the network to classify images of digits. The simple



Train Residual Network on CIFAR-10

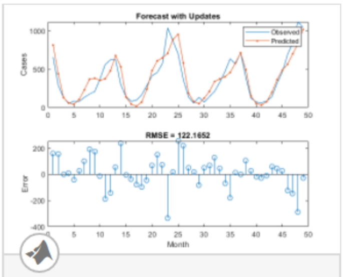
Create a deep learning neural network with residual connections and train it on CIFAR-10 data. Residual connections is a popular

Deep Learning with Time Series and Sequence Data:



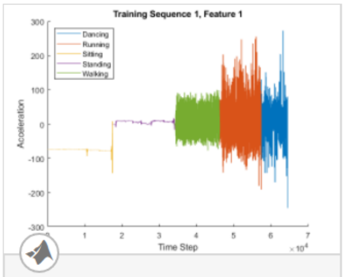
Sequence Classification Using Deep Learning

Classify sequence data using a long short-term memory (LSTM) network.



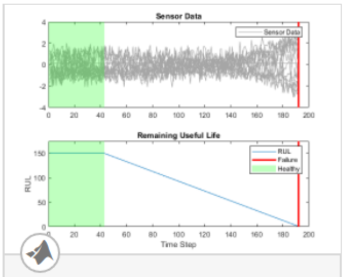
Time Series Forecasting Using Deep Learning

Forecast time series data using a long short-term memory (LSTM) network.



Sequence-to-Sequence Classification Using Deep Learning

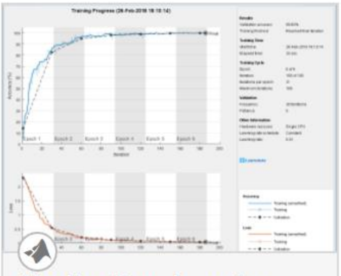
Classify each time step of sequence data using a long short-term memory (LSTM) network.



Sequence-to-Sequence Regression Using Deep Learning

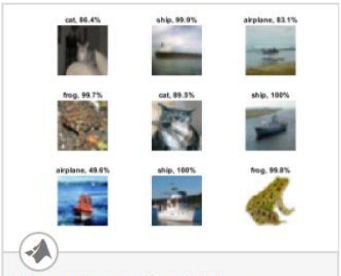
Predict the remaining useful life (RUL) of engines by using deep learning.

Deep Learning Tuning and Visualization:



Monitor Deep Learning Training Progress

When you train networks for deep learning, it is often useful to monitor the training progress. By plotting various metrics during training, you

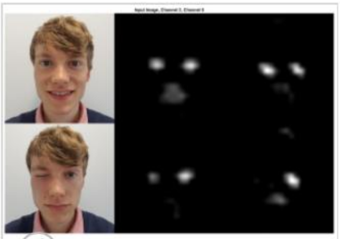


Deep Learning Using Bayesian Optimization

Apply Bayesian optimization to deep learning and find optimal network parameters and training options for convolutional neural networks.

Deep Dream Images Using AlexNet

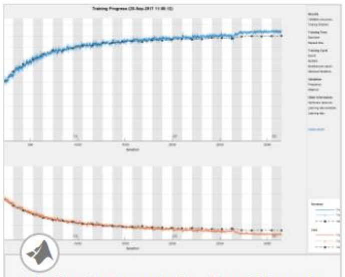
Generate images using deepDreamImage with the pretrained convolutional neural network AlexNet.



Visualize Activations of a Convolutional Neural Network

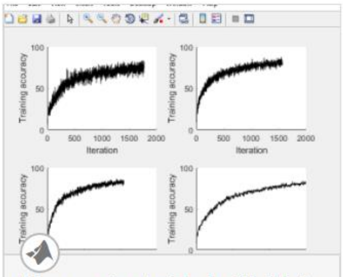
Feed an image to a convolutional neural network and display the activations of different layers of the network. Examine the activations

Deep Learning in Parallel and in the Cloud



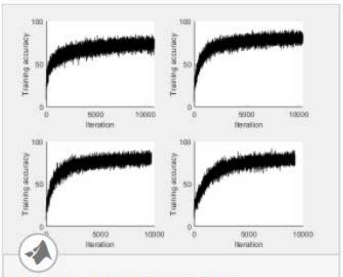
Train Network in the Cloud Using Built-in Parallel Support

Train a convolutional neural network on CIFAR-10 using MATLAB's built-in support for parallel training. Deep Learning training often takes hours



Use parfor to Train Multiple Deep Learning Networks

Use a parfor loop to perform a parameter sweep on a training option. Deep Learning training often takes hours or days, and searching



Use parfeval to Train Multiple Deep Learning Networks

Use parfeval for a parameter sweep on the depth of the network architecture. Deep Learning training often takes hours or days, and

Upload Deep Learning Data to the Cloud

Upload your data to an Amazon S3 bucket. Before you can perform deep learning training in the cloud, you need to upload your data to the

Deep Learning with Computer Vision:

Semantic Segmentation Using Deep Learning

Train a semantic segmentation network using deep learning.

Object Detection Using Deep Learning

Train an object detector using deep learning and R-CNN (Regions with Convolutional Neural Networks).

Object Detection Using Faster R-CNN Deep Learning

Train an object detector using a deep learning technique named Faster R-CNN (Regions with Convolutional Neural Networks).

Deep Learning with Image Processing:

Remove Noise from Color Image Using Pretrained Neural Network

Remove Gaussian noise from an RGB image. Convert the noisy image to the L*a*b* color space, and remove noise on the luminance

High-Resolution Results Using Bicubic Interpolation (Left) vs. VDSR (Right)

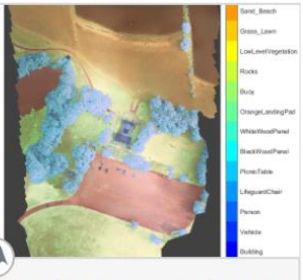
Single Image Super-Resolution Using Deep Learning

Train a Very-Deep Super-Resolution (VDSR) neural network, then use a VDSR network to estimate a high-resolution image from a single low-

rom JPEG-Compressed Images with Quality Factor: 50, 20 and 10 (left

JPEG Image Deblocking Using Deep Learning

Train a denoising convolutional neural network (DnCNN), then use the network to reduce JPEG compression artifacts in an image.



Semantic Segmentation of Multispectral Images Using Deep Learning

Train a U-Net convolutional neural network to perform semantic segmentation of a multispectral image with seven channels: three

Deep Learning with Automated Driving:

Train a Deep Learning Vehicle Detector

Train a vision-based vehicle detector using deep learning.

Create Occupancy Grid Using Monocular Camera and Semantic…

Estimate free space and create an occupancy grid using semantic segmentation and deep learning. You then use this occupancy grid to

Deep Learning Deployment:

Code Generation for Deep Learning Networks

Demonstrates code generation for an image classification application that uses deep learning. It uses the codegen command to generate a

Object Detection

Generate CUDA® code from a SeriesNetwork object created for YOLO architecture trained for classifying the PASCAL dataset.

Lane Detection Optimized With GPU Coder

Generate CUDA® code from a deep learning network, which is represented by a SeriesNetwork object. The SeriesNetwork in this

Traffic Sign Detection and Recognition

Demonstrates how to generate CUDA® MEX code for a traffic sign detection and recognition application, that uses deep learning.

MatCNN Toolbox (from Matlab Central - LINK):

- MatCNN is a MATLAB toolbox implementing Convolutional Neural Networks (CNNs) for computer vision applications. It is simple, efficient (integrating MATLAB GPU support), and can run and learn state-of-the-art CNNs, similar to the ones achieving top scores in the ImageNet challenge. Several example CNNs are included to classify and encode images.
- An important feature of MatCNN is making available the CNN building blocks as easy-to-use MATLAB commands. This allows prototyping new CNN architectures and learning algorithms as well as recycling fast convolution code for sliding window object detection and other applications.
- MatCNN is developed by a team of computer vision scientists in Oxford and other research institutions.
- Requires: Parallel Computing Toolbox

Implementing CNNs using Google's TensorFlow: MNIST Tutorial

Variational Autoencoders ("Generative" Models): Tutorial by Jaan Altosaar