Convolutional networks are NNs that use convolution in place of general matrix multiplication in at least one of their layers
The function s(t), which applies a weighted average favoring more recent steps, is called a convolution

$$s(t) = \int x(a)w(t-a)da$$

Denoted as the following typically (using an asterisk) $s(t) = (x * w)(t)$
w needs to be a valid probability distribution in order for output to be a weighted avg; furthermore, w needs to be 0 for all negative arguments, or else it will look into the future
The first arg, the function x, is the in put, second argument is the function w, the kernel, and the final arg is the feature map
w does not necessarily need to be a weight function

Discrete convolutions, which are more practical, are defined as follows

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

Sensor reads every second or so, for example

Input and kernels are multidimensional arrays, called tensors
Assume kernel and input functions are 0 everywhere except for the stored data points
This lets us implement the infinite summation as a summation over a finite # of elements

Convolutions can used over more than 1 axis at a time - for example, on an image:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

Since convolution is commutative, we can write

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n).$$

i and j correspond to t, a particular instant
The latter formula is easier to implement since there is less variation in the range of valid values of m and n when using the image matrix

This property arises since we have flipped the kernel relative to the input; originally, as m increases, the index into the input increases and the index into the kernel decreases
The only reason to flip the kernel is to obtain the commutative property and write proofs

Cross correlation is the same as convolution but does not flip the kernel

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n).$$

Regardless of flipping or not, the algorithm learns the right values in the corresponding spot

Convolution is used with other functions, and the order does matter (regardless of whether convolution uses flipping or not)

Discrete convolution can be viewed as multiplication with a matrix with constraints
Toeplitz matrix - each row is constrained to be equal to the row above it shifted right by 1 in one dimension
Doubly blocked circulant matrix corresponds to convolution in two dimensions
Convolution corresponds to a sparse matrix, since the kernel is much smaller than the input image, and any neural net algorithm which makes predictions independent of matrix structure should be able to work with convolution

- Making entries zero helps this space independency

Note that the global operation (not each indiviudal kernel application) is the Toeplitz matrix

Motivation
Convolution uses 3 important things: parameter sharing, sparse sharing, equivariant representation; it also lets us work with variable size inputs
Typical NNs have connections between each output and each input; CNNs use sparse connectivity or sparse weights to make sure only important info is kept
This can reduce runtime from $O(m \times n)$ to $O(k \times n)$, where k is typically multiple orders of magnitude smaller than m
Units in the deeper layer may indirectly interact with a larger portion of the input from simple building blocks using sparse connections

In CNNs, parameters are shared or tied; parameter sharing / tying in this case is a result of the same kernel being applied to every input location, rather than having a separate set of params
This reduces storage (but not runtime) of the model to k parameters, making it practically insignificant compared to m x n, making it more effective than dense matrix multiplication

Convolution also has a property of equivariance, where g(f(x)) = f(g(x)); if g is any function which translates the input, the convolution is equivariant to g (the output changes in the same way if the input does)
If $I' = g(I)$ and $I'(x, y) = I(x - 1, y)$., then shifting the pixel one to the right and applying convolution is the same as applying convolution and shifting it one to the right
This means when processing timeseries, convolution creates a map which shows when different features appear in the data - if the same thing happens many time steps later, this is reflected in the output
Sometimes, we may want to focus on specific locations in the image
Convolution is not naturally equivariant to some operations, like rotation / scaling

Pooling
First two stages - several convolutions in parallel to produce feature maps, then they go through an activation function (detector stage), then they are run through a pooling function

Pooling replaces the output of a net / grid at a certain location with a summary statistic of nearby output; some examples are max pooling, average, L^2 norm of rectangular neighborhood, or weighted average based on distance from a certain point

Pooling helps make representations become approximately invariant to changes in the input, which is important when determining if a feature is present in the data, as opposed to where

- Detecting a face - do not need the precise locations of the eyes
- Infinitely strong prior that the function a layer learns must be translation invariant

Pooling over the outputs of separately parametrized convolutions lets the features learn which transformations to become invariant to

It is possible to use fewer pooling units than detector units by reporting summary statistics for pooling regions spaced k pixels apart, giving the next layer k times fewer inputs to process
Many times, pooling is essential for handling varying size inputs (e.g breaking up different sizes images into 4 quadrants)

Can learn a pooling structure, or run a clustering algorithm to find different regions to pool over

Convolution and Pooling as Infinitely Strong Prior
Strong prior - low entropy (e.g Gaussian with low variance) plays an active role in determining where the parameters end up
Infinite prior gives certain parameter values 0, regardless of how much support data gives

Convolutional net - prior says weights must be identical to its neighbor, but shifted in space
Weights must be 0, except for small receptive fields assigned to its units
Says that the function the layer should learn only contains local interactions
Pooling gives an infinitely strong prior of invariance to small translation
Convolution and pooling can cause underfitting - they are only useful when the assumptions made by their priors are valid (e.g translational invariance is important)

Models can be permutation invariant and must discover topology through learning, and models can be hardcoded to have spatial relationships

Variants of the Basic Convolution Function
Since we want many features at many locations, we use many convolutions / kernels in parallel
3D tensors due to the number of channels - one index specifies RGB, other two specify grid coords (multichannel convolution)
Multichannel convolutions are only commutative if each op has the same # of outputs as inputs

Suppose we have a tensor K with $K_{i,j,k,l}$ giving connection strength between unit in channel i of the output and unit in channel j in the input, with an offset of rows and cols of k and l (K specifies general inner product convolution weights)

$V_{i,j,k}$ - channel i at row j and column k; the output Z has the same format as V
If Z is produced by convolving K across V without flipping K, then

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}$$

(Linear algebra indexing starts at 1)
A downsampled convolution function will skip s pixels to reduce computational cost

$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} \left[V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n} \right].$$

s is the stride of the downsampled convolution

Zero padding the input V makes it wider
Without this, the width of the representation would shrink by one pixel less than the kernel width at each layer
Zero padding lets us control kernel width and output size independently, and prevents us from needing to use a small kernel (which would reduce the power of the network)

Valid convolution - no zero padding, output shrinks at each layer
If kernel has width k, output will have width of m - k + 1
If the spatial dimension is finally reduced to 1x1, additional layers cannot be considered "convolutional"
Same convolution - just enough zero padding such that the width of the next layer is the same
Input pixels near the border influence less output pixels than those in the middle, causing border pixels to become underrepresented
Full convolution - enough zeroes are added so each pixel can be visited k times in each direction (both left and right), resulting in an output of m + k - 1
Output pixels near the edge are a function of fewer pixels than the center, making it hard to learn a kernel which can fit to both of these conditions at once

If we want to use locally connected layers rather than convolution, give every connection its own weight specified by a 6D tensor W
The indices into W are i, the output channel, j, the output row, k, the output col, l, the input channel, m, the row offset in the input, and n, the col offset in the input
Linear part of a locally connected layer is given by

$$Z_{i,j,k} = \sum_{l,m,n} \left[V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n} \right].$$

This is called unshared convolution, since it is similar to regular convolution but wts not tied
Useful for when we know that a feature is a small part of the space, but does not appear everywhere (e.g mouth is always in bottom half)

You can constrain each output channel i to only be a subset of the input channels l
This gives fewer parameters, less computation, and does not decrease the # of hidden units

Tiled convolution - in between local and regular convolution, where a set of kernels is learned as we rotate through space; immediately neighboring locations will have different filters, but since each individual location does not have its own filter, the overall memory for storing the params increases by a factor of the size of the set of kernels

Let k be a 6-D tensor where two dimensions correspond to different locations in the output
Output locations cycle through a set of t different choices of kernel stack in each direction

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n,j\%t+1,k\%t+1},$$

If t = output width, this is the same as local convolution

If detector units of locally connected and tiled convolution layers learn to detect different versions of the same underlying features, max-pooled units become translation invariant
Convolutional layers are hard coded to be translationally invariant

Gradient must be calculated w/ respect to the kernel, but this cannot be done using the convolution function if the stride > 1
Convolution can be described as matrix multiplication, if the input tensor is flattened to a vector, where the matrix involved is a function of the convolution kernel
Each element of the kernel is copied to several elements of the sparse matrix

Multiplication of the transpose of the matrix defined by convolution is an operation needed to backpropagate error derivatives through a convolution layer

Generally, the input gradient operation requires a third operation (besides convolution, and backprop from outputs to weights) to be implemented
Size of output of transpose depends on zero padding policy, stride of forward prop, and size of the forward prop output map
Transpose operation must be explicitly told size of the input, since many input sizes of forward prop can result in the same dimensional output map

Strided convolution of kernel stack K applied to an image V w/ stride s = c(K, V, s)

c outputs Z, which after backprop, the tensor $G_{i,j,k} = \frac{\partial}{\partial Z_{i,j,k}} J(\mathbf{V}, \mathbf{K}).$ is backpropagated
Compute the derivatives with respect to the kernel weights using a function

$$g(\mathbf{G}, \mathbf{V}, s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k,l}} J(\mathbf{V}, \mathbf{K}) = \sum_{m,n} G_{i,m,n} V_{j,(m-1)\times s+k,(n-1)\times s+l}.$$

K is the particular connection between two layers - sum up (the partial derivative of the error w/ respect to each row and column element multiplied by distance to the unit given by the row and column offset), and do this for each row to each unit
Note that this gives the partial derivative for each individual instance of an offset (e.g for each particular k and l) -> this does not yield the overall gradient for all offsets

Gradient w/ respect to V needs to be calcualted to backpropagate the error further down

$$h(\mathbf{K}, \mathbf{G}, s)_{i,j,k} = \frac{\partial}{\partial V_{i,j,k}} J(\mathbf{V}, \mathbf{K})$$
$$= \sum_{\substack{l,m \\ \text{s.t.} \\ (l-1)\times s+m=j}} \sum_{\substack{n,p \\ \text{s.t.} \\ (n-1)\times s+p=k}} \sum_{q} K_{q,i,m,p} G_{q,l,n}.$$

This goes through all offsets, for every row col pair, where each kernel map w/ index q corresponds to an output channel

PCA copies an input x to a reconstruction r using $W^T Wx$
General autoencoders use multiplication by the transpose of the weight matrix, similar to PCA
Use function h to perform the transpose of the convolution op to make the function convolutional
$\mathbf{R} = h(\mathbf{K}, \mathbf{H}, s).$ where H is the hidden units
Gradient w.r.t R is a tensor E
To train the decoder, we need the gradient w.r.t K, g(H, E, s); to train the encoder, we need it w.r.t H, given by c(K, E, s) <- c and g are functions of the derivatives of R

For tiled convolution, the same biases are shared where the same kernels are used; for locally connected layers, each one typically has its own bias
For regular convolutional layer, all neurons in the input map have the same bias when producing each individual output map after the kernel operation

- May need to adjust the biases so that they are nonuniform (edge pixels which only receive zero padded inputs may need a stronger bias)

Structured Outputs
Neural nets can output a high dimensional, structured tensor
S an have $S_{i,j,k}$ which corresponds to the probability that pixel (j, k) of network belongs to class i
One issue is the output dimension has smaller dimensionality than the input dimension; this may be fixed by removing the stride, emiting a low res grid of labels, or using a unit stride

Another strategy is to produce an initial guess, and then refine it with interactions between neigboring pixels; repeating this step corresponds to using the same convolutions at each stage, sharing weights between the last layers of the deep net

- This makes the sequence of computations by the conv. layers a recurrent convnet

Once a prediction is made, these predictions can be further processed to get a segmentation of these images into regions - the general idea is neighbornig pixels have the same labels, specifically described by graphical models, which may be approximated by the convnet itself

Data Types
Convnets can process data w/ varying spatial extent, which cannot be done by ANNs
The same convolution kernel creates a different size doubly block circulant matrix for each input

Sometimes the output may have a variable size
Other times, when we need a fixed size, we must do something like have a pooling layer which scales proportionally ot the input layer
Processing variable size inputs is subject to the constrant that the inputs must be related over the width / height / time / new dimension

Efficient Convolution Algorithms
Convolution is euqivalent to converting both the input and kernel to the frequency domain using Fourier transformation, performing point wise mult. of the two signals,and converting back to time domain using inverse Fourier <- this may be faster than discrete convolution in some cases

When a d dimensional kernel can be expressed as an outer product of d vectors, each one corresponding to a dimension, the kernel is separable
When the kernel is separable, naive convolution is equivalent to composing d one-dimensional convolutions with each of these vectors
Having multiple one-dimensional convolutions is more efficient than 1 multi-dimensional
Kernel takes fewer parameters to represent when represented as vectors

- If it is w elements wide in each dimension, naive multidimensional convolution requires $O(w^d)$ while separable requires $O(w * d)$ runtime and storage space

Research being done to improve approximations of convolutions of networks

Random or Unsupervised Features
Output layer is inexpensive due to the small number of features after pooling
Using features which are not trained in a supervised fashion can speed up grad. descent

Three ways to obtain convolution kernels w/o supervised training:

- Random initialization
- Design them by hand (e.g setting them to detect edges at a scale)
- Unsupervised criterion, such as k-means

Learning the features w/ an unsupervised criterion allows them to be determined independently from the classifier at the top - once the features are extracted once, it is like a new training set for the layer at the top

Convolution layers followed by pooling become translation invariant and frequency selective even when assigned random weights
Supervised greedy pretraining - train the first layer in isolation, move on to the second one, and so on
We can take this one step further with convnets - instead of training each convolution layer one step at a time, we can train a model of a small patch using unsupervised learning
This lets us not use convolution during the training process, causing a low computational cost

Neuroscientific Basis
Cats respond to patterns of light in the early layers, but no obvious patterns

V1, as part of the brain, is a spatial map with a 2D structure - light arrives at the lower half of the retina, affecting the corresponding half of the V1; features being defined in 2D maps in convnets capture this property
V1 contains simple cells, corresponding to receptive fields
V1 contains complex cells, using information from multiple distant sources - pooling helps close these distances, so they may be processed by kernels in later layers

In the human brain, moving beyond the V1, the later neurons are invariant to certain changes
A convnet is very similar to a IT when a human gaze is interrupted and the first 100 ms of feedforward process is observed

Intermediate computations of neurons in the brain probably uses very different activation functions; furthermore, simple cells and complex cells may be the same cell responding as needed by the situation

Time delay neural networks are 1D convnets applied to time series, and these were the first to experience backprop

In a biological neural network, we put an electrode in front of a neuron, display examples of white noise, and record how these cause the neurons to activative; then, fit a linear model to get an approximation of a weights - this is reverse correlation

Reverse correlation shows V1 cells have weights described by Gabor functions, describing the weight at a 2D point in the image
The image is a function of 2D coordinates I(x, y) - a simple cell samples the image at a set of locations and applies weights which are functions of location w(x, y)

$$s(I) = \sum_{x \in \mathbb{X}} \sum_{y \in \mathbb{Y}} w(x, y) I(x, y).$$

w(x, y) takes the form of a Gabor function:

$$w(x, y; \alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau) = \alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right) \cos(f x' + \phi)$$

$x' = (x - x_0)\cos(\tau) + (y - y_0)\sin(\tau)$ and $y' = -(x - x_0)\sin(\tau) + (y - y_0)\cos(\tau).$

α, βx, βy, f, φ, x0, y0, and τ are parameters which control the properties of Gabor functions
x_0, y_0, and t define a coordinate system; translate and rotate x and y to form x' and y'
Simple cell responds to image features centered at (x_0, y_0) with increasing brightness as we move along a line rotated tau radians from the horizontal

As a function of x' and y', w responds to changes in brightness as we move along the x axis:
Two important factors - a Gaussian function and cosine function
$\alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right)$ ensures the simple cell only responds to values near where x' and y' are both 0, or when they are near the center of the cell's receptive fields

α adjusts the magnitude of the cell's response, while B_x and B_y control how quickly the receptive field falls off across position

Cosine factor $\cos(fx' + \phi)$ controls how much the simple cell responds to changing brightness along the x axis <- f controls frequency and phi controls phase offset

Complex cell computes the L^2 norm of the 2D vector containing simple cells' responses

$$c(I) = \sqrt{s_0(I)^2 + s_1(I)^2}.$$

s_0 and s_1 form a quadrature pair when s_1 is one quarter cycle out of phase with s_0
A complex cell defined in this way responds when the Gaussian weighted image $I(x, y)\exp(-\beta_x x'^2 - \beta_y y'^2)$ contains a high amplitude sine wave with frequency f in direction tau near (x_0, y_0), regardless of the phase offset
Basically, the complex cell is invariant to small translations of the image in direction tau, or when negating / inverting the image